

QL▪ Assembler

Contents

Page

1 Introduction	A3
1.1 Back-up Copy	A4
1.2 Notation used in this manual	A5
2 How to run the assembler	A7
3 Assembler inputs and outputs	A14
3.1 Control inputs	A14
3.2 Source inputs	A14
3.3 Library input	A16
3.4 Screen output	A18
3.5 Source listing	A20
3.6 Symbol table listing	A21
3.7 Object code output	A21
4 Listing outputs	A22
4.1 Source listing	A24
4.2 Symbol table listing	A28

Appendix A	Bibliography	A30
-------------------	--------------	-----

Appendix B	Source language	A33
-------------------	-----------------	-----

B.1	Lexical analysis	A33
-----	------------------	-----

B.2	Source language line format	A37
-----	-----------------------------	-----

B.3	Expressions	A40
-----	-------------	-----

B.4	Addressing modes	A44
-----	------------------	-----

B.5	Instructions	A51
-----	--------------	-----

B.6	Assembler directives	A65
-----	----------------------	-----

Appendix C	Error and warning messages	A77
-------------------	-------------------------------	-----

C.1	Error messages	A77
-----	----------------	-----

C.2	Warning messages	A84
-----	------------------	-----

C.2	Operating system errors	A86
-----	-------------------------	-----

The QL▪Assembler manual is Copyright © 1984,
GST Computer Systems Limited. The document is
not to be reproduced in whole or in part without
prior permission from GST Computer Systems Ltd.,
91 High Street, Longstanton, Cambridge CB4 5BS.

1 Introduction

This manual tells you how to use QL Assembler produced by GST Computer Systems for the Sinclair QL.

It tells you:

- (a) how to load and run the assembler
- (b) what inputs the assembler takes and what outputs it produces
- (c) how the assembler language instructions should be coded
- (d) what assembler directives are available, what they do, and how to code them.

It does not:

- (a) include a detailed description of the instruction set of the Motorola MC68000 processor family (which includes the 68008 as used in the QL) for which you will need additional documentation
- (b) tell you how to talk to QDOS, the QL's operating system, for which you will have to consult the QDOS Software Developers Guide
- (c) teach programming in general
- (d) teach assembler programming or 68000 programming in particular.

Appendix A contains a list of some other publications which you may find helpful.

1.1 Back-up Copy

You are advised to make a back-up copy with the spare cartridge using the clone program provided with the Assembler. Place the blank cartridge into drive 1 and the master cartridge into drive 2, then **LRUN MDV2-CLONE** and follow the instructions on the screen.

1.2 Notation used in this manual

This section describes the notation used throughout the manual to describe syntax of assembler source, as well as other items.

- = means that the expression on the right defines the meaning of the item on the left, and can be read as "is"
- < > angle brackets containing a lower-case name represent a named item which is itself made up from simpler items, such as
`<decimal number>`
- | a vertical bar indicates a choice and can be read as "or is"
- [] square brackets indicate an optional piece of syntax that may appear 0 or 1 times
- { } curly brackets indicate a repeated piece of syntax that may appear 0 or more times
- ... is used informally to denote an obvious range of choices, as in:
`<digit> = 0|1|...|8|9`

Other symbols stand for themselves.

Example

```
<binary number> =  
    %<binary digit>  
    {<binary digit>}  
<binary digit> = 0|1
```

means that a binary number is a '%' sign followed by a binary digit, followed by any number of further binary digits, where a binary digit is the character '0' or the character '1'. Some examples of binary numbers are %0, %1010101100, %00000000000000.

Some of the special symbols used in the syntax notation also occur in the assembler source input and the common sense of the reader is relied on to distinguish these, as in for example:

```
<operator> = ...|<<|...
```

2 How to run the assembler

The assembler or editor may be run using the QL's `EXEC` or `EXEC_W` from Super Basic or by booting from microdrive directly. It may be run from any microdrive (or other storage medium) and the files it uses may be on any combination of storage media.

Use control-C to change to the assembler input line.

Typical command lines

`EXEC MDV2_ASM` – start assembler from drive 2.

`EXEC MDV1_ED` – start editor from drive 1.

The first thing the assembler does is to ask you for a window specification. Normally you can just press `ENTER` in which case the assembler will use the device `CON_` (with default position and size) for its screen output and keyboard input. If however you have something else on the screen already which you do not wish to destroy you may type in a complete `CON_` specification (see the QL manual) and the assembler will use the window you specify instead.

It will say hello and ask you to type a command line. This command line identifies all the files and devices that the assembler will use and can also be used to specify various options.

See section 3 of this manual for a description of all the various files and devices that the assembler can use.

The format of the command line is:

```
<source> [<listing>  
  [<binary>]] {<option>}
```

where:

```
<option> =  
-NOLIST |  
-ERRORS [<listing>] |  
-LIST [<listing>] |  
  
-NOBIN |  
-BIN [<binary>] |  
  
-NOSYM | -SYM |  
  
-LIBRARY <library> |  
-LIB <library>
```

(the options may be in upper or lower case and case is not significant)

```
<source> = <file name>  
file name of assembler source
```

```
<listing> = <file name>  
file name for listing output
```

<binary> = <file name>

file name for binary output

<library> = <file name>

file name for library input

The options have the following meanings:

- NOLIST** do not generate any listing output
- ERRORS** generate a listing of error messages and erroneous lines only; if the option is followed by a **<file name>** then this is the name of the **<listing>** output and the positional **<listing>** parameter, if coded, is not used; the **-ERRORS** option also sets the **-NOSYM** option
- LIST** generate a full listing; if the option is followed by a **<file name>** then this is the name of the **<listing>** output and the positional **<listing>** parameter, if coded, is not used
- NOBIN** do not generate any binary output
- BIN** generate binary output; if the option is followed by a **<file name>** then this is the name of the **<binary>** output file and the positional **<binary>** parameter, if coded, is not used

- NOSYM** do not generate a symbol table listing; this is the default if **-ERRORS** is coded
- SYM** generate a symbol table listing; this is the default if **-LIST** is coded or if no listing options are coded; if both **-SYM** and **-NOLIST** are both coded then the **-SYM** does nothing
- LIBRARY** (or **-LIB**) the **-LIBRARY** option must be followed by a `<file name>` and specifies a file containing a precompiled library to be included in the assembly

Where conflicting options are given the last one coded takes effect. For example, if:

```
-LIST mdv1_fred -NOLIST  
-ERRORS
```

is coded then an errors-only listing will be sent to **MDV1_FRED**, and if:

```
-SYM -ERRORS
```

is coded then no symbol table output will be generated.

The minimum command line just consists of the name of the input source file. In this case a full listing with symbol table is generated (i.e. the default is `-LIST -SYM`) to the file whose name is constructed from the `<source> <file name>` as described below. Also by default a binary output file is generated (i.e. the default is `-BIN`) to the file whose name is constructed from the `<source> <file name>` as described below.

The `<source> <file name>` is examined: if its last four characters (after converting to upper case) are not `"_ASM"` then `"_ASM"` is appended to the given name to make the name of the actual source file used.

The name of the `<listing>` file may be given positionally as the second parameter, or may be specified explicitly after a `-ERRORS` or `-LIST` option, or may be allowed to default. If no `<listing> <file name>` is given in a `-ERRORS` or `-LIST` option and no `-NOLIST` option has been coded then the assembler constructs the `<listing> <file name>` by taking the `<source> <file name>`, as adjusted, and replacing the `"_ASM"` with `"_LIST"`.

The name of the <binary> file may be given positionally as the third parameter, or may be specified explicitly after a -BIN option, or may be allowed to default. If no <binary> <file name> is given in a -BIN option and no -NOBIN option has been coded then the assembler constructs the <binary> <file name> by taking the <source> <file name>, as adjusted, and replacing the "_ASM" with "_BIN".

Examples:

MDV1_FRED

assemble MDV1_FRED_ASM, put a full listing with symbol table listing in MDV1_FRED_LIST, and put the binary in MDV1_FRED_BIN.

MDV1_FRED SER1 -nobin

assemble MDV1_FRED_ASM, print the listing as it is produced, and don't generate any binary

MDV1_FRED -ERRORS -BIN

Mdv2_fred_bin

assemble MDV1_FRED_ASM, send an error listing only with no symbol table to

MDV1_FRED_LIST, and put the binary in MDV2_FRED_BIN (note that coding mdv2_fred would not have achieved this)

```
MDV1_FRED ser1 mdv2_fred_bin  
-errors -SYM -library  
MDV1_qdos_lib
```

assemble MDV1_FRED_ASM, print an error listing plus symbol table directly, put the binary in MDV2_FRED_BIN and include the precompiled file MDV1_QDOS_LIB in the output binary.

When the assembly has finished, and if there have been no operating system errors, the assembler will not go away but will repeat the prompt asking for a command line. You can now do another assembly without having to reload the assembler. When you have done all the assemblies that you want you may reply to this prompt with an empty command line and the assembler will terminate.

3 Assembler inputs and outputs

This chapter describes all the input and output files and devices that the assembler can use.

3.1 Control inputs

Control information for the assembler is supplied by the user typing a command line on the keyboard. The command line is described in section 2 above and specifies where all the other input and output files and devices are.

3.2 Source inputs

The assembler assembles one main source file. This may direct the assembler, using **INCLUDE** directives, to read other source files.

When assembling large and complicated programs it is normal to put no real code at all in the main source file which will just contain **INCLUDE** directives naming the other source files. For example:

TITLE A large complicated assembly

*

* Start with the QDOS parameter file, then the
* parameter file for my program
*

INCLUDE mdv1_qdos_in

INCLUDE mdv1_myparms_in

*

* Now the main code to be assembled: this is
* rather large so it is split into two separate files
*

INCLUDE MDV2_PROG1_IN

INCLUDE MDV2_PROG2_IN

*

* Finally, the -LIBRARY facility is being used
* to include a library of useful subroutines; the
* declaration file for the library must be
* INCLUDED last
*

INCLUDE MDV1_LIBRARY_IN

END

The filename of the main source file must end in
_ASM, or the assembler will not be able to find it.

It is recommended that filenames of INCLUDED
files end in _IN, but this is not a requirement and
you can call them anything you like.

3.3 Library input

The assembler's library mechanism allows you to include in your program a previously assembled binary file containing useful subroutines or other code. The program being assembled may refer to labels within the library, but the library must be self-contained and cannot refer to labels elsewhere in the program.

To use a library you must make reference to two files.

The first file is a set of symbol definitions, in normal assembler source format, which you must **INCLUDE** at the **end** of your source program. This file causes your references to library symbols to be resolved so that your code can be assembled.

The second file is a binary file containing the code of the library routines. You must present this to the assembler by giving its name in the **-LIBRARY** option on the command line.

If you manage to leave out one of these two files, or use a definition file that is not compatible with the binary file, then undefined chaos will result.

You can build your own libraries as follows:

- (a) write the code
- (b) assemble it: the output from the assembler is now the library binary file
- (c) build a definitions file from the symbol table listing resulting from the assembly: for each symbol in the library which you wish to be able to access from programs, write a line:

```
symbol EQU *+offset
```

where `offset` is the value printed for the symbol on the listing.

You can extend libraries in the obvious way as the assembly in step (b) above can itself use a library.

3.4 Screen output

The assembler writes a certain amount of information to the screen to let the user know what is happening. This includes a "hello" message, a "finished" message and the request to type the command line.

A summary of the number of errors and warnings generated is written to the screen together with a summary of the amount of memory used. This memory size excludes the memory occupied by the code of the assembler itself (about 17k) and the assembler's initial data space (about 6k).

You can get a good idea of how complicated your assemblies are and whether you are likely to run out of memory by watching the memory use figure. On a standard 128k QL it is possible to assemble a source file which occupies more or less a whole microdrive tape as long as nothing much else is using memory at the same time.

If you do several assemblies in one go (without reloading the assembler) then the assembler will reuse any memory it has obtained from the operating system but will not release any memory until it terminates completely. This means, for example, that if you do a very large assembly followed by a very small assembly there will be no more free memory in the QL during the small assembly than there was during the large assembly.

The assembler also tells you when it is starting to read the source input for the first time and when it is starting to read the source input for the second time. The second pass can be expected to take a lot longer than the first pass if listings and/or binary output are wanted. The symbol table listing is produced after the summary messages are displayed, so if you are assembling a large program it will be an appreciable time after the summary messages are displayed before the assembler finishes completely.

3.5 Source listing

An optional source listing will be generated, showing the source input and the code that has been generated.

The listings provided are controlled both by options on the command line (see section 2 above) and by directives coded in the source program (see appendix B below).

If the `-NOLIST` option is given then there will be no listing output from the assembler. Under all other circumstances a file or device will be used to produce a listing.

If the filename for the listing output is generated automatically by the assembler it will end in `_LIST`. It is recommended that listing files, when stored on microdrive, always have filenames ending in `_LIST`, but this is not a requirement and you can call them anything you like.

Listings can be printed directly as they are generated (using `SER1` or `SER2` or some add-on printer device) or can be sent to the screen (using `CON_`) as an alternative to sending them to microdrive.

3.6 Symbol table listing

A symbol table listing will be produced if both the `-LIST` and `-SYM` options are in effect.

The symbol table listing will be added to the end of the source listing, starting on a new page.

3.7 Object code output

The assembler produces a binary file that may be executed using the QL's `EXEC` or `EXEC_W` command, as desired.

See the QDOS Software Developers Guide for further details.

The size of the data space that is allocated to a program may be specified by the user in a `DATA` directive (see appendix B); if no `DATA` directive is given then the size of the data space will be 4096 bytes.

4 Listing outputs

There are two listings produced by the assembler: the source listing and the symbol table listing.

Each line of listing produced can be up to 132 characters long (excluding the terminating newline); in particular each title line is 132 characters long. Some printers cannot be made to print 132 characters to a line so the **PAGEWID** directive (q.v.) is provided to specify the actual width of the printer. Any line longer than **PAGEWID** characters will be overflowed onto the following line, and these overflows will be taken account of when determining whether a page is full.

The listing output is paginated with the total page length defined by the user in a **PAGELN** directive (q.v.) or allowed to default. To obtain essentially unpaginated output the user may set **PAGELN** to a very large number, in which case only one title will be printed at the beginning of the listing, and form feeds will be included at the start and end of the listing and between the source and symbol table listings only.

The format of each printed page is:

<heading>
<blank>
<title>
<blank>
<blank>
<listing>
<form feed>

where:

<blank>	is a blank line (i.e. a line feed character)
<heading>	is a line containing the name and version of the assembler, the name of the source file being assembled, the page number, and the time and date
<title>	is the <title string> given on the relevant TITLE directive; if no relevant TITLE directive has been coded then this line is <blank>
<listing>	consists of (PAGELEN - 14) lines of listing of whatever format is appropriate (source listing or symbol table listing)
<form feed>	is the ASCII form feed character and appears immediately after the line feed which terminates the last line (if any) of <listing>

4.1 Source listing

Note that if the `-ERRORS` option has been requested then not all source lines are listed: only lines containing errors are listed, together with the error messages.

Each line of listed source code has the following format:

Columns	Field contents	Format
1-4	line number	4-digit decimal
5	(blank)	
6	section number	1-digit hex
8-15	location counter	8-digit hex
16	(blank)	
17-28	generated code	up to 12 digits hex
29	(blank)	
30-132	source line	as coded, truncated to fit

Source line numbers start at 1 for the first line in the (main) source file and are incremented by 1 for each source line processed regardless of the file from which it came and regardless of whether the line is listed or not.

The section number is zero for instructions and data assembled into section zero. It is left blank when absolute addresses (such as those generated under the influence of an `OFFSET` directive) are being displayed.

For instructions and data definition directives the location counter field contains the address which would be assigned to a label defined on that source line; note that this is not necessarily the same as the value of the location counter after the previous line has been processed. For other directives containing expressions whose value is likely to be of interest to the user (e.g. `OFFSET`, `EQU`) the value of the expression is printed in the location counter field or the code field, as appropriate. If there is nothing useful that can be printed in this field then it is left blank.

The generated code field contains up to 6 bytes of code generated by an instruction or a data definition directive (`DC` or `DCB`). If an instruction generates more than 6 bytes of code then a second listing line is used to display the rest of it; this second listing line is blank apart from the generated code field (and possibly some error flags). Code in excess of 6 bytes generated by `DC` or `DCB` directives is not printed; if you want to see it you should code several separate `DC` or `DCB` directives.

The length of the listing line is in all cases limited to 132 characters, any excess (probably comment) being truncated.

Error and warning messages are interspersed with the source listing; each message follows the listing of the line to which it refers. If a line has errors or warnings it is followed by a line containing a vertical bar character (|) below the part of the source line giving offence. The format of the messages is:

```
***** ERROR xx — line nnnn —  
      mmmm — <message>  
**** WARNING xx — line nnnn —  
      mmmm — <message>
```

where *xx* is the error number, *nnnn* is the line number of the line containing the error, *mmmm* is the line number of the line containing the previous error (0 if none) to allow the user to chain through all the error messages to make sure none have been missed, and *<message>* is a helpful message saying what is wrong. There are separate chains for error and warning messages.

The line giving rise to an error or warning is always listed, regardless of the state of any *LIST* or *NOLIST* directives. Thus the listing generated by *-ERRORS* is more or less the same as the listing generated by *-LIST* if *NOLIST* directives are in force throughout.

If there is no **END** directive a special warning message is printed relating to this at the end of the assembly; the line number in this warning message is one greater than the number of the last line in the input file.

At the end of the assembly a summary of the number of errors and warnings generated is output both to the listing, if there is one, and to the screen.

4.2 Symbol table listing

The symbol table listing is a sorted list of each user-defined symbol with its type, value and line number of the line on which it was first defined.

The listing is sorted alphabetically on symbol name, with ASCII collating sequence for non-alphabetic characters. It is printed in a single column.

The symbol table listing for each symbol contains the following fields:

Columns	Field contents	Format
1-8	symbol	up to 8 characters
9	(blank)	
10-13	symbol type	see below
14	(blank)	
15	section number	0 or X or R, see below
16	(blank)	
17-24	value	8-digit hex
25	(blank)	
26-29	line number	4-digit decimal

The type field contains:

MULT	if the symbol is multiply defined; the assembler will use the first definition and print error messages for subsequent ones
blank	ordinary labels

The section number field contains:

blank	symbol is absolute
0	symbol is simple relocatable and lives in section 0
X	symbol is complex relocatable
R	symbol is a register list defined by a REG directive

If the symbol is undefined then the section number and value fields will contain the word 'undefined'.

The line number field contains the line number of the first line in which the symbol was defined: for an undefined symbol it is left blank.

Appendix A – Bibliography

QDOS Software Developers Guide.

This manual describes the facilities of QDOS that are available to the assembler programmer and tells you how to call them.

You will need this book to write programs for the QL. It does not attempt to teach 68000 programming.

Available from Sinclair Research Limited.

Programming the MC68000 by Tim King and Brian Knight, Addison-Wesley

This is an excellent book which teaches assembler programming on the 68000 and also contains a complete description of the 68000's instruction set. It is suitable for the first-time assembler programmer although you should do some programming in another language, such as SuperBasic, before using assembler. This book is also very valuable to the experienced assembler programmer who has not used a 68000 before as it points out many of the common errors and pitfalls which usually cause trouble for the newcomer to the 68000.

Available by post from GST Computer Systems Limited, 91 High Street, Longstanton, Cambridge at £8.95 including postage and packing.

M68000 16/32 Bit Microprocessor Programmer's Reference Manual

This is the Motorola handbook for the 68000 (reference number M68000UM). It contains definitions of the 68000 instruction set (as does the King and Knight book) and in addition contains more low-level information, such as details of the binary code for each instruction and some hardware information.

Available by post from GST Computer Systems Limited, 91 High Street, Longstanton, Cambridge at £8.95 including postage and packing.

Appendix B – Source language

This appendix defines the source language accepted by the assembler. It does not specify the details of the Motorola 68000 instruction set and a manual for the 68000 itself must be consulted for this information.

B.1 Lexical analysis

This section defines the way in which characters are combined to make tokens. The notation used is described in section 1 above.

Generally a line of assembler source is divided into the traditional four fields of label, operation, operand and comment, the fields being separated by spaces.

Thus spaces are significant in this language, apart from just terminating symbols.

As a special case a line containing an asterisk in column one consists entirely of comment and is treated as a blank line.

Any syntactic token is terminated either by the first character which cannot form part of that token or by end of line.

```

<syntactic token>
  = <white space> |
    <symbol> |
    <number> |
    <string> |
    <newline> |
    << | >> |
    !|#|&|(|)|*|+|,|-|/|:
    (where <newline> is a line feed character)
<white space>
  = <space> {<space> }
    (where <space> is the ASCII space character)
<symbol>
  = <start symbol>
    {<rest symbol>}
<start symbol>
  = <letter>|.
<rest symbol>
  = <letter>| <digit> |$|.|-
<letter>
  = a|b|...|y|z
    |A|B|...|Y|Z

```

note that (outside strings) whether a letter is upper or lower case is not significant

note that a symbol can be any length but only the first eight characters are significant

```

<number>
= <binary number>|
  <octal number>|
  <decimal number> |
  <hex number>
<binary number>
= %<binary digit>
  {<binary digit>}
<octal number>
= @<octal digit>
  {<octal digit>}
<decimal number>
= <digit> {<digit>}
<hex number>
= $<hex digit>
  {<hex digit>}
<binary digit>
= 0|1
<octal digit>
= 0|1|...|6|7
<digit>
= 0|1|...|8|9
<hex digit>
= <digit>
  |a|...|f|A|...|F

```

```

<string>
= '<stringchar>
  {<stringchar>}'

```

where a <stringchar> is any ASCII character except a line feed, a control character, or a single quote ' ; in addition a <stringchar> may be two adjacent single quotes which allows a single quote to be coded inside a string

There are two types of `<symbol>` used by the assembler. `<symbol>`s appearing in the operation field are "operation type symbols" and those appearing in the operand field are "operand type symbols". These two sets of `<symbol>`s are quite separate and there is no confusion (except in the mind of the programmer) between the same name used in both places. Thus you can have user-defined labels with the same names as instructions and directives, if you really want to.

There are special forms of strings used by the `INCLUDE` and `TITLE` directives which allow the user to omit the enclosing quotes:

```
<file name>  
= <string>| {<non space  
character>}
```

i.e. a `<file name>` is either enclosed in quotes or is terminated by a space or end of line

```
<title string>  
= {<character>}
```

i.e. a `<title string>` is terminated by end of line

B.2 Source language line format

This section defines the various forms which a source line can take.

A source line consists of between 0 and 132 characters (excluding the line feed character).

Basically a source line consists of the following four fields:

label	(optional, but depends on operation)
operation	(optional)
operand	(depends on operation)
comment	(optional)

A source line can be blank (including consisting entirely of comment as defined above) in which case it is ignored for all purposes other than those connected with output listings: a blank line is assigned a line number, is printed on the listing, and its position may affect the operation of the title directive.

B.2.1 The label field

A line contains a label field if it starts with one of the following sequences of tokens:

```
<symbol><white space>  
<symbol>:  
<white space><symbol>:
```

i.e. a label starting in column 1 may be followed by `<white space>` or a colon, but a label starting further along the line must be terminated by a colon.

Such a sequence at the start of a line is referred to elsewhere in this appendix as a `<Label>`.

If a line contains a label and contains nothing after the label then the label is defined with the current value of the current location counter; otherwise the meaning of the label depends on the operation field.

B.2.2 The operation field

The operation field follows the (optional) label field and its syntax is:

[<white space>]<symbol>

The symbol is one of:
an assembler directive
a 68000 instruction

B.2.3 The operand field

The syntax of the operand field depends on the operation. <white space> terminates the operand.

The syntax of each format of the operand field is described below when the operation is defined.

B.2.4 The comment field

When enough of the rest of the line has been processed to satisfy the operation (for the majority of operations this is up to the first <white space> beyond the start of the operand field) anything left on the line is deemed to be comment and ignored.

B.3 Expressions

Expressions are constructed from:

unary operators:	<code>+</code> , <code>-</code>
binary operators:	<code>+</code> , <code>-</code> , <code>/</code> , <code>*</code> , <code>>></code> , <code><<</code> , <code>&</code> , <code>!</code>
parentheses:	<code>(</code> , <code>)</code>
operands:	<code><symbol></code> , <code><number></code> , <code>*</code> , <code><string></code>

`<string>`s used in expressions must be four characters long or shorter. The value of a `<string>` consists of the ASCII values of the characters right-justified in the normal 32-bit value. Thus, for example, the two expressions

`'a'*256+'b'` and `'ab'`

have the same value. (Note that the `DC` directive can use longer strings with different evaluation rules.)

The character `*` used as an expression operand has the same value as a `<label>` defined on the line in which the `*` is used would have.

The syntax of an expression is then:

```

<expr>
= <symbol> |
  <number> | * |
  <string> |
  (<expr>) |
  + <expr> | <expr> |
  <expr> <binaryop>
  <expr>
<binaryop>
= +|-|*|<<
  | >> | & | !

```

The operators have the following meanings:

unary + the value of the operand is unchanged
 unary - the value of the operand is negated

Note that all operands are regarded as 32 bit values; these values are obtained by extending the original operand on the left with zeroes (all operands are originally positive except that symbols can be defined to have negative values, in which case they will already be 32 bit negative numbers). Likewise all intermediate and final results from expressions are calculated as 32 bit values and are truncated as necessary according to context just before being used.

binary +	addition
binary -	subtraction
*	multiplication
/	division: the result is truncated towards zero
<<	shift left: the left operand is shifted to the left by the number of bits specified by the right operand, which should be an absolute value between 0 and 32 inclusive otherwise the result is undefined; vacated bits at the right hand end are filled with zeroes
>>	shift right: as for shift left but the operand is shifted right
&	bitwise logical AND
!	bitwise logical OR

The order of evaluation of expressions is as follows:

- parenthesised expressions are evaluated first (in the natural way)
- operators are evaluated according to priority; the order of priority is (highest first):
 - unary +, -
 - <<, >>
 - &, !
 - *, /
 - binary +, -
- operators of the same precedence at the same nesting level of parentheses are evaluated from left to right.

Symbols may be absolute or relocatable. Numbers and strings are absolute; the current location counter (*) is relocatable. The only operators which may act on relocatable symbols or relocatable subexpressions are unary + and - and binary + and -.

When an expression has been fully evaluated it is one of:

- (a) absolute: the final value is independent of the start of section 0
- (b) simple relocatable: the final value is an offset from the start of section 0
- (c) complex relocatable: the final value involves some other multiple of the start of section 0

B.4 Addressing modes

This section defines all addressing modes that can be coded as instruction operands. For a definition of what these addressing modes actually do consult a manual for the Motorola 68000.

B.4.1 Addressing mode syntax

A number of symbols are reserved and have special meaning when used in operands: these are names of various registers.

D0 to D7	data registers also the symbols D0.W, D0.L etc.
A0 to A7	address registers also the symbols A0.W, A0.L etc.
SP	synonym for A7 also the symbols SP.W, SP.L
USP	user stack pointer
CCR	condition code register (low 8 bits of SR)
SR	status register
PC	program counter

The syntax of instruction operands is developed below, preceded by a few general definitions.

```

<areg> =
    A0 | ... | A7 | SP
<dreg> =
    D0 | ... | D7
<iereg> =
    <areg> | <dreg> |
    A0.W | ... | A7.W | SP.W | D0.W
    | ... | D7.W |
    A0.L | ... | A7.L | SP.L | D0.L
    | ... | D7.L
<multireg>
    = <range> {/<range>}
<range>
    = <areg> | <dreg> | <areg>-
    <areg> |
    <dreg>-<dreg>

```

(where the registers in an individual range must be in increasing register order, e.g. D0-D3 is valid and A4-A2 is not valid)

The addressing modes which are called (by Motorola) "effective address" and which can be coded (or at least a subset of them) in any instruction which has a general effective address as an operand are:

<code><ea> =</code>	
<code><dreg> </code>	D register direct
<code><areg> </code>	A register direct
<code>(<areg>) </code>	register indirect
<code>(<areg>)+ </code>	postincrement
<code>-(<areg>) </code>	predecrement
<code><expr></code>	
<code>(<areg>) </code>	indirect with displacement
<code><expr></code>	
<code>(<areg> ,</code>	
<code><i reg>) </code>	indirect with index
<code><expr> </code>	absolute short
<code><expr> </code>	absolute long
<code><expr> </code>	PC relative
<code><expr></code>	
<code>(PC) </code>	PC relative
<code><expr></code>	
<code>(PC ,</code>	
<code><i reg>) </code>	PC with index
<code>#<expr></code>	immediate

Note that the syntax `<expr>` means either PC with displacement addressing or either form of absolute addressing, and this ambiguity is resolved according to the semantics of the `<expr>`. See below for details.

Also the operand `<dreg>` (e.g.) could be either a register direct addressing mode or a `<multireg>` and hence a multiple register specification: the assembler is capable of deciding what is meant depending on the instruction being assembled.

B.4.2 Interpretation of addressing modes

Basically all references which involve relocatable destinations must be PC-relative for the code to be position independent which is a requirement for running under QDOS. This means that references to labels more than 32k bytes away will fail, and the programmer must find some other means of reaching the destination.

All forms of the effective address are coded exactly as meant apart from

`<expr>`

which can mean an absolute short address, an absolute long address or a PC-relative address.

If the value of the `<expr>` is absolute the assembler will generate an absolute short address if possible, otherwise it will generate an absolute long address.

If the value of the <expr> is relocatable the assembler will try to generate a PC-relative address. This will fail if the destination is too far away or if the effective address is required to be 'alterable'; in either case an error message will be produced and the programmer must find some other way of writing the program.

Forward references which are undefined at the time of meeting the symbol are assumed to be simple relocatable. If the programmer wishes to reference an absolute address this can only be done by coding a number or by coding a symbol which has previously been equated to a number. For example:

```
                MOVE.B    #$80,SCREEN
                .....
SCREEN EQU      $18063
```

is not legal and will generate an error, whereas:

```
                JMP      FRED
                ....
FRED
```

is legal and will generate a PC-relative addressing mode.

An immediate operand #<expr> where the <expr> is not absolute will probably generate wrong code as the assembler does not know where the code will be loaded and executed and is unable to add the necessary relocation base(s). Therefore the assembler will generate warning messages if a relocatable <expr> is used as an immediate operand.

B.4.3 Branch instructions

The branch instructions (Bcc, BSR) can use either an 8-bit PC-relative displacement or a 16-bit displacement; the assembler will correctly choose the most efficient option for a backwards reference but needs some help with forward references. The default option is to generate a long (16-bit) displacement.

These branch instructions can have an explicit extent coded of .S (short) meaning that an 8 bit displacement is to be used or .L (long) meaning that a 16 bit displacement is to be used, for example:

BNE.S FRED FRED is nearby

B.5 Instructions

This section lists all the 68000 instruction mnemonics, describes how the various modifiers are coded, and defines the operand syntax of each instruction. Note however that for precise details of the actual addressing modes etc. legal for each instruction a manual for the Motorola 68000 should be consulted.

An instruction may optionally have a `<Label>`. Before any code for an instruction is generated the current location counter is advanced to an even address if not already even and it is this adjusted address that is assigned to the `<symbol>` in the `<Label>`.

B.5.1 Instruction mnemonic format

The operation field of a source line containing a machine instruction is simply a `<symbol>`. However there is some flexibility allowed in the coding of mnemonics as there are some generic mnemonics that relate to a group of instructions, the actual instruction wanted being chosen by the assembler depending on the operands coded.

Instructions which may operate on operands of different lengths must have the length of the operand coded as part of the `<symbol>`: this takes the form of `" . B "`, `" . W "` or `" . L "` as the last two characters of the `<symbol>` depending on whether the operand length is byte, word or long. If a length is required and no length is coded the assembler will assume `. W` and will print a warning message.

Instructions which may only take a single operand length may optionally have the length coded as above.

The branch instructions may optionally have `" . S "` or `" . L "` coded as the last two characters of the `<symbol>` to indicate the displacement size as described at B.4.3 above.

Examples:

<code>MOVE . L</code>	an instruction with an operand length coded
<code>BEQ . S</code>	an instruction with an extent coded
<code>JSR</code>	an instruction with no extra bits
<code>MOVE . L D0 , A0</code>	automatically generates <code>MOVEA . L</code>
<code>MOVE . L #2 , D3</code>	automatically generates <code>MOVEQ . L</code>

B.5.2 Data movement instructions

The various forms of the **MOVE** instruction are used to move data between registers and/or memory. These are:

MOVE<length> <ea>,<ea>

which is the generic instruction, and will generate one of the following if necessary:

MOVEA<length> <ea>,<areg>

MOVEQ[.L] #<expr>,<dreg>

Note that both **MOVEA** and **MOVEQ** can be coded explicitly if desired. Note also that the assembler will only convert a **MOVE** to a **MOVEQ** if the length is specified as ".L".

Various other special forms of the **MOVE** instruction are always coded as **MOVE** (they have no specific mnemonic) but they all operate on a single length of operand and the operand length is optional. These are:

MOVE[.W] <ea>,CCR

MOVE[.W] <ea>,SR

MOVE[.W] SR,<ea>

MOVE[.L] <areg>,USP

MOVE[.L] USP,<areg>

The **MOVEM** and **MOVEP** instructions are also involved with data movement but are not generated automatically by the assembler from the **MOVE** mnemonic. Their syntax is:

MOVEM<length> <multireg>,
 <ea>

MOVEM<length> <ea>,
 <multireg>

MOVEP<length> <dreg>,<expr>(<areg>)

MOVEP<length> <expr>(<areg>),
 <dreg>

The other data movement instructions are:

EXG[.L] <reg>,<reg>

where

<reg> =

<areg> | <dreg>

LEA[.L] <ea>,<areg>

PEA[.L] <ea>

SWAP[.W] <dreg>

B.5.3 Arithmetic instructions

In a similar way to the **MOVE** instruction, the **ADD**, **CMP** and **SUB** mnemonics are generic and will generate **ADDA**, **ADDI**, **ADDQ**, **CMPA**, **CMPI**, **CMPM**, **SUBA**, **SUBI**, **SUBQ** if necessary; again, the explicit forms can be coded if desired.

ADD <length>	<ea>, <ea>
CMP <length>	<ea>, <ea>
SUB <length>	<ea>, <ea>

ADDA <length>	<ea>, <areg>
ADDI <length>	#<expr>, <ea>
ADDQ <length>	#<expr>, <ea>

CMPA <length>	<ea>, <areg>
CMPI <length>	#<expr>, <ea>
CMPM <length>	(<areg>)+, (<areg>)+

SUBA <length>	<ea>, <areg>
SUBI <length>	#<expr>, <ea>
SUBQ <length>	#<expr>, <ea>

Additional (binary) arithmetic instructions are:

ADDX <length>	<dreg>, <dreg>
ADDX <length>	-(<areg>), -(<areg>)

CLR <length>	<ea>
---------------------	------

DIVS[.W]	<ea>,<dreg>
DIVU[.W]	<ea>,<dreg>
EXT<length>	<dreg>
MULS[.W]	<ea>,<dreg>
MULU[.W]	<ea>,<dreg>
NEG<length>	<ea>
NEGX<length>	<ea>
SUBX<length>	<dreg>,<dreg>
SUBX<length>	-(<areg>)
	-(<areg>)
TST<length>	<ea>

The binary coded decimal instructions are written as follows:

ABCD[.B]	<dreg>,<dreg>
ABCD[.B]	-(<areg>),
	-(<areg>)
NBCD[.B]	<ea>
SBCD[.B]	<dreg>,<dreg>
SBCD[.B]	-(<areg>),
	-(<areg>)

B.5.4 Logical operations

AND, EOR, OR are generic mnemonics that will generate ANDI, EORI, ORI as necessary:

AND<length>	<ea>,<dreg>
AND<length>	<dreg>,<ea>
AND<length>	#<expr>,<ea>
ANDI<length>	#<expr>,<ea>

EOR<length>	<dreg>,<ea>
EOR<length>	#<expr>,<ea>
EORI<length>	#<expr>,<ea>

NOT<length>	<ea>
-------------	------

OR<length>	<ea>,<dreg>
OR<length>	<dreg>,<ea>
OR<length>	#<expr>,<ea>
ORI<length>	#<expr>,<ea>

There are special forms of the **ANDI**, **EORI** and **ORI** instructions which operate on the status register.

AND.B #<expr>,SR
AND.W #<expr>,SR
AND[.B] #<expr>,CCR

ANDI.B #<expr>,SR
ANDI.W #<expr>,SR
ANDI[.B] #<expr>,CCR

EOR.B #<expr>,SR
EOR.W #<expr>,SR
EOR[.B] #<expr>,CCR

EORI.B #<expr>,SR
EORI.W #<expr>,SR
EORI[.B] #<expr>,CCR

OR.B #<expr>,SR
OR.W #<expr>,SR
OR[.B] #<expr>,CCR

ORI.B #<expr>,SR
ORI.W #<expr>,SR
ORI[.B] #<expr>,CCR

B.5.5 Shift operations

ASL<length> <dreg>,<dreg>
ASL<length> #<expr>,<dreg>
ASL[.W] <ea>

ASR<length> <dreg>,<dreg>
ASR<length> #<expr>,<dreg>
ASR[.W] <ea>

LSL<length> <dreg>,<dreg>
LSL<length> #<expr>,<dreg>
LSL[.W] <ea>

LSR<length> <dreg>,<dreg>
LSR<length> #<expr>,<dreg>
LSR[.W] <ea>

ROL<length> <dreg>,<dreg>
ROL<length> #<expr>,<dreg>
ROL[.W] <ea>

ROR<length> <dreg>,<dreg>
ROR<length> #<expr>,<dreg>
ROR[.W] <ea>

ROXL<length> <dreg>,<dreg>
ROXL<length> #<expr>,<dreg>
ROXL[.W] <ea>

ROXR<length> <dreg>,<dreg>
ROXR<length> #<expr>,<dreg>
ROXR[.W] <ea>

B.5.6 Bit operations

The length specification is optional on these instructions as the length must be long if the `<ea>` is a `<dreg>` and must be byte if the `<ea>` is anything else.

BCHG[<code><length></code>]	<code><dreg></code> , <code><ea></code>
BCHG[<code><length></code>]	<code>#<expr></code> , <code><ea></code>
BCLR[<code><length></code>]	<code><dreg></code> , <code><ea></code>
BCLR[<code><length></code>]	<code>#<expr></code> , <code><ea></code>
BSET[<code><length></code>]	<code><dreg></code> , <code><ea></code>
BSET[<code><length></code>]	<code>#<expr></code> , <code><ea></code>
BTST[<code><length></code>]	<code><dreg></code> , <code><ea></code>
BTST[<code><length></code>]	<code>#<expr></code> , <code><ea></code>

B.5.7 Branch instructions

The branch instructions may optionally have an extent (`.S` or `.L`) coded as described at B.4.3. above.

B`<cc>[<extent>] <expr>`

where:

<code><cc>=</code>	<code>CC</code>	<code>CS</code>	<code>EQ</code>	<code>GE</code>	<code>GT</code>	<code>HI</code>
	<code>LE</code>	<code>LS</code>	<code>LT</code>	<code>MI</code>		
	<code>NE</code>	<code>PL</code>	<code>VC</code>	<code>VS</code>	<code>HS</code>	<code>LO</code>

`<extent> = .S | .L`

The unconditional branch instruction is:

BRA`[<extent>] <expr>`

and is in fact a version of the conditional branch instruction that means "branch regardless of the condition codes".

The branch to subroutine instruction is:

BSR`[<extent>] <expr>`

B.5.8 Trap instructions

Grouped here are those instructions whose main purpose is to generate traps, either conditionally or unconditionally.

CHK[.W] <ea>,<dreg>

TRAP #<expr>

TRAPV

B.5.9 The DBcc instruction

This instruction is a looping primitive; it tests the condition codes as does the Bcc instruction but also allows the conditions "always true" and "always false" to be tested.

DB<dbcc>[.W] <dreg>,<expr>

where:

<dbcc> = <cc> | T | F | RA

RA is a synonym for F, meaning branch regardless of the condition codes; thus the instruction **DBRA** loops without testing conditions other than the value of the loop counter.

B.5.10 Jump instructions

The jump instructions are an unconditional jump and a subroutine call:

JMP <ea>

JSR <ea>

See section B.4.2 for a definition of how the assembler interprets <expr> as an <ea>, as that paragraph is particularly relevant to these two instructions.

B.5.11 Stack frame management

LINK <areg>, #<expr>

UNLK <areg>

B.5.12 Odds and ends

NOP
RESET
RTE
RTR
RTS
TAS[.B] <ea>
STOP #<expr>

The S c c instruction has the same set of conditions as DB c c but not the RA synonym:

S<scc>[.B] <ea>

where:

<scc> = <cc> | T | F

B.6 Assembler directives

Assembler directives are instructions to the assembler and, with the exception of **DC** and **DCB**, do not directly generate any code. The directives provided are summarised below.

The following directives must not have labels:

INCLUDE	read another source file
SECTION	relocatable program section
OFFSET	define offset symbols
DATA	specify data space
END	end of program

The following directives require labels:

EQU	assign value to symbol
REG	define a register list

The following directives may optionally have labels:

DC	define constants
DS	reserve storage
DCB	define constant block

The following are listing control directives and must not have labels:

PAGE	start new listing page
PAGEWID	define width of page
PAGELN	define length of page
LIST	switch listing on
NOLIST	switch listing off
TITLE	define title for listing

B.6.1 INCLUDE – read another source file

This directive causes the named file to be read as if it were present in the original source file in place of the **INCLUDE** directive. **INCLUDE** directives may be nested to three levels.

The syntax of an **INCLUDE** directive is:

INCLUDE <file name>

where <file name> (with optional surrounding quotes) is the normal syntax of a file name for QDOS.

B.6.2 SECTION – start relocatable section

This directive defines the relocation base to be used for subsequent code generation. The only section implemented is section 0.

No SECTION directive need be coded unless OFFSET is used, in which case a SECTION directive must separate sequences of OFFSET definitions from following code.

Any number of SECTION directives may be present.

The syntax of the SECTION directive is:

SECTION <expr>

where the expression must be absolute, contain no forward references, and have the value zero.

B.6.3 OFFSET – define offset symbols

The **OFFSET** directive provides a means for symbols to be defined as offsets from a given point: this is particularly useful for defining field names for data structures.

The **<expr>** given in an **OFFSET** directive must be absolute and must not contain forward references or external references. The value of the **<expr>** is the initial value of a dummy location counter which can then be used to define labels on following **DS** directives.

The syntax of the **OFFSET** directive is:

OFFSET <expr>

Between an **OFFSET** directive and a following **OFFSET** or **SECTION** (or **END**) directive the following are not allowed:

DC, DCB, instructions.

B.6.4 END – end of program

The **END** directive defines the end of the source input; if there is anything else in the file on subsequent lines then this will be ignored by the assembler.

The syntax of the end statement is:

END

A68

B.6.5 EQU – assign value to symbol

Syntax:

`<label> EQU <expr>`

The `<expr>` is evaluated and the value is assigned to the `<symbol>` given in the `<label>`

The `<expr>` may not include references to any symbol which has not yet been defined.

The value of the defined symbol is absolute, simple relocatable or complex relocatable depending on the type of the `<expr>`.

B.6.6 REG – define register list

Syntax:

`<label> REG <multireg>`

The `<symbol>` given in the `<label>` is defined to refer to the register list given in `<multireg>` and may be used in **MOVEM** instructions only.

The purpose of this directive is to allow a symbol to be defined which represents a register list pushed at the start of a subroutine so that the same list of registers can be popped at the end of the subroutine without the risks involved in writing the list out twice.

B.6.7 DC – define constants

This directive defines constants in memory. Memory is reserved and the values of the constants given are stored in this memory. This facility is intended to allow constants and tables to be created.

Syntax:

```
[<label>] DC<length>  
    <constant> {,<constant>}
```

where:

```
<constant> =  
    <expr> | <string>
```

If a <constant> consists of a single string and no other operators or operands then it is left justified in as many bytes, words or long words (depending on whether <length> is .B, .W or .L) as necessary, with the last word or long word padded with zero bytes as necessary. In this case the <string> can be of any (non-zero) length; there is no restriction as there is with <string>s that form part of <expr>s.

This leads to a rather strange feature that:

```
DC.L 'a'
```

causes the character to be left-justified whereas

```
DC.L 'a' + 0
```

is an `<expr>` and so causes the character to be right-justified. (Note that other 68000 assemblers have even stranger features in this area.)

In the case of `DC.W` and `DC.L` the current location counter is advanced to a word boundary if necessary, and the optional `<label>` is defined with this adjusted value. This the code fragments:

```
FRED DC.W ....
```

and

```
FRED  
    DC.W ...
```

do not necessarily have the same effect as the second could result in `FRED` having an odd value depending on earlier use of `DC.B`, `DS.B` or `DCB.B`.

Expressions given as operands of `DC` directives must be absolute.

No more than six bytes of code generated by a `DC` are printed on the listing; if all generated bytes are required then the constants must be coded on more separate `DC` directives.

B.6.8 DS – reserve storage

This directive reserves memory locations. The memory contents are undefined. The directive is used to define offsets in conjunction with the `OFFSET` directive and to leave "holes" in data generated by `DC` and `DCB`; it is also of use in ensuring that the current location counter has an even value.

Syntax:

```
[<label>] DS<length> <expr>
```

If the length is `.W` or `.L` the current location counter (which can be a dummy location counter initiated by `OFFSET`) is advanced to a word boundary if necessary. The (optional) `<label>` is assigned the value of the adjusted location counter.

The `<expr>` must be absolute and contain no forward references.

`DS.B` reserves `<expr>` bytes, `DS.W` reserves `<expr>` words and `DS.L` reserves `<expr>` long words.

`<expr>` may have the value zero in which case `DS.W` and `DS.L` ensure that the location counter is on an even boundary, and the optional `<label>` is defined.

B.6.9 DCB – define constant block

The directive:

```
[<label>] DCB<length>  
           <expr>,<expr>
```

causes the assembler to generate a block of bytes, words or longs depending on whether <length> is .B, .W or .L.

If the length is .W or .L the current location counter is advanced to a word boundary if necessary. The (optional) <label> is assigned the value of the adjusted location counter.

The first <expr> must be absolute and contain no forward references and is the number of storage units (bytes, words or longs) to be initialised, and the second <expr> is the value to be stored in each of these storage units.

The second <expr> should be absolute.

B.6.10 PAGE – start new listing page

The directive

PAGE

causes the next line of the listing to appear at the top of the next page. The PAGE directive itself is not listed.

B.6.11 PAGEWID – define width of page

The directive

PAGEWID <expr>

defines the width of the printed output to be <expr> characters. The <expr> must be absolute and contain no forward references and must be between 72 and 132 inclusive. If no **PAGEWID** directive is present the default is 132 characters.

B.6.12 PAGELEN – define length of page

The directive

PAGELEN <expr>

defines the length of each listing page to be <expr> lines. The <expr> must be absolute and must contain no forward references. The value given is the physical length of the paper; rather fewer lines of assembler source are actually listed on each page. If no **PAGELEN** directive is present the default is 66 lines.

B.6.13 LIST – switch listing on

The directive

LIST

restarts listing that was suppressed by a previous **NOLIST** directive. The **LIST** directive itself is not listed.

B.6.14 NOLIST – switch listing off

The directive

NOLIST

suppresses listing until a **LIST** directive is encountered. The **NOLIST** directive itself is not listed.

B.6.15 TITLE – define title for listing

The directive

TITLE <title string>

causes the <title string> to be printed at the top of each subsequent page of listing. If a title is wanted on the first page of the listing then the **TITLE** directive should appear before any source line which would get listed. The **TITLE** directive itself is not listed.

B.6.16 DATA – define size of data space

The directive

DATA <expr>

defines the size of the data space that will be allocated to the program when it is executed by QDOS. The <expr> gives the number of bytes to be reserved.

The expression must be absolute and contain no forward references.

If several **DATA** directives are coded the last one takes effect.

If no **DATA** directives are coded then 4096 bytes of data space will be allocated to the program.

Appendix C – Error and warning messages

This appendix lists the error and warning messages which can be produced by the assembler in numerical order.

C.1 Error messages

00 – unknown instruction/directive

An unknown symbol has been used where an instruction or directive is expected in the operation field.

01 – illegal line after OFFSET

Instructions and directives which generate code (DC, DCB) are not allowed in the dummy section defined by the OFFSET directive. Return to SECTION 0 before instructions or data.

02 – syntax error in instruction field

The operation field does not contain a <symbol>.

03 – redefined symbol

The symbol has already been defined earlier in the assembly. The first definition of the symbol will be used; further definitions will just produce this error message.

04 – phasing error

This is an assembler internal error – it should only happen if the source file has changed between pass 1 of the assembler and pass 2.

05 – missing operand

The instruction requires two operands, and only one has been coded.

06 – syntax error

The line contains a syntax error which has left the assembler with very little idea of what was meant.

07 – syntax error in expression or operand

The assembler is expecting an expression or other instruction operand but does not understand what it has found.

08 – multireg, cannot mix Dreg & Areg

Data registers and address registers may not be combined in a range: eg D3–A4 is illegal.

09 – multireg, bad sequence

The registers in a range must be in increasing order – eg D5–D2 is illegal.

0A – unmatched open bracket

There are too many open brackets in the expression: unmatched open brackets are “closed” at the end of the expression.

0B – unmatched close brackets

There are too many close brackets in the expression: unmatched close brackets are ignored.

0C – expression too complicated

An expression is limited to five levels of nested brackets. Certain combinations of operators can cause this error with fewer brackets – eg when low priority operators are followed by high priority operators.

0D – expression: string too long

When a string is used as a term in an expression it may be up to four characters long.

0E – value stack underflow

This is an internal assembler error which should never occur.

0F – invalid character

Some characters such as " ? \ ^ = have no meaning to the assembler. They may only be used within strings. The character is ignored.

10 – invalid shift operator

The characters "<" and ">" are only legal as pairs in shift operators: ">>" and "<<".

11 – no digits in number

A number is expected (eg after "\$" or "%") but no digits are present.

12 – number overflow

The number is too large and will not fit in 32 bits.

13 – string terminator missing

A string must be terminated by a quote character.

14 – relocatable value not allowed here

Some addressing modes and directives require absolute values.

15 – multiply overflow in expression

A multiply overflow error occurred while evaluating an expression.

16 – divide by 0 or divide underflow

A divide error occurred during evaluation of an expression.

18 – -ve value illegal

Some directives (eg **D S**) can accept a zero or positive number, but a negative value is illegal.

19 – value must be +ve nonzero

Some instructions or directives require a positive, nonzero, value (eg the number of elements for **D C B**).

1 A – value out of range

This is a general purpose message for any value out of range in instructions or directives. The actual value range depends on context – reread the description of the instruction or directive involved.

1 D – size not allowed on directive

Most directives do not accept a size extension: the only ones that do allow a size are **DC**, **DCB** & **DS**.

1 E – invalid size

The size specified on the instruction or directive is not legal.

1 F – size .B illegal for Areg

Byte operations on address registers are not allowed.

2 0 – label illegal on this directive

Many directives (eg **INCLUDE**, **SECTION**, **LIST**, **PAGE**) do not accept a label.

2 1 – too many errors

If a line has more than ten errors or warnings, only the first ten are printed, followed by this message.

2 2 – invalid operand(s) for this instruction

The operand(s) specified are not valid for the instruction. Check the rules for the instruction you are using in a 68000 manual. If one of the operands to the instruction is an "effective address" this error can mean that the actual addressing mode specified is not legal.

The assembler will try to point the error flag (the vertical bar character) at the invalid operand, but as the assembler may not even know (in the case of a generic mnemonic) which instruction you meant it will get this wrong sometimes.

23 – undefined symbol

The symbol has not been defined in the assembly.

24 – forward reference not allowed here

Many directives do not allow a forward reference.

25 – short branch out of range

BRA . S (or some other **B r a n c h . S**) has been coded but the destination is more than 128 bytes away.

26 – long branch out of range

The destination of a long branch must be within 32k.

27 – value must be simple relocatable

The expression should be simple relocatable: absolute or complex values are illegal (e.g. in the destination of a branch instruction).

28 – value must not be complex

Absolute and simple relocatable expressions can generally be used as addresses but a complex relocatable value is illegal.

29 – this directive must have a label

EQU and **REG** require a label.

2 A – unable to generate position independent code here

Normally if a label or expression is used to specify an address in an instruction, a PC-relative addressing mode is generated to produce position independent code. This is not an alterable addressing mode, so this error message is generated when an alterable addressing mode is required.

2 B – short branch to next instruction – NOP generated

A short branch to the next instruction is not a legal 68000 opcode. The assembler generates a NOP instruction in this case.

C.2 Warning messages

4 0 – size missing, W assumed

No size was specified on an index register.

4 1 – size missing, W assumed

The instruction or directive can have more than one size, but no size was specified.

4 2 – multiply defined register

A register has been multiply defined in a multiregister sequence (eg `A0/D1/D0-D3` has `D1` multiply defined).

4 3 – decimal number goes negative

A decimal number has a value between `$80000000` and `$FFFFFFFF`. This is a perfectly valid number with which to do unsigned arithmetic, but it is an overflow if the programmer was intending to use it for signed arithmetic. As the assembler does not know what the programmer wants to do with the number it produces this warning.

4 4 – nonzero SECTION not implemented

Implementation restriction: only one relocatable section is supported.

4 5 – value will be sign extended to 32 bits

in `MOVEQ` the expression is between `$80` and `$FF` so it will be sign-extended to a 32-bit negative value.

4 6 – nonstandard use of this instruction

This warning is printed when an instruction is used in a nonstandard manner which may be a bug (eg `LINK` with a positive displacement).

4 7 – branch could be short

A forwards branch or a branch with an explicit `.L` is within 128 bytes range and could be a short branch.

4 8 – END directive missing

An `END` directive is expected at the end of the assembly, but end-of-file was found instead.

C.3 Operating system errors

When the assembler gets an error code from QDOS it usually gives up completely, first displaying a message relating to the error on the screen.

Most QDOS errors relate to particular input or output files or devices and the file or device name involved is displayed as part of the message wherever possible.

In the case of a serious error (such as bad microdrive tape) affecting an input source file the assembler does not however tell you which of the various source (e.g. `INCLUDE`d) files is involved.

If the assembler is run with `EXEC_W` the error code is passed back to the `EXEC_W` command which will display another error message.