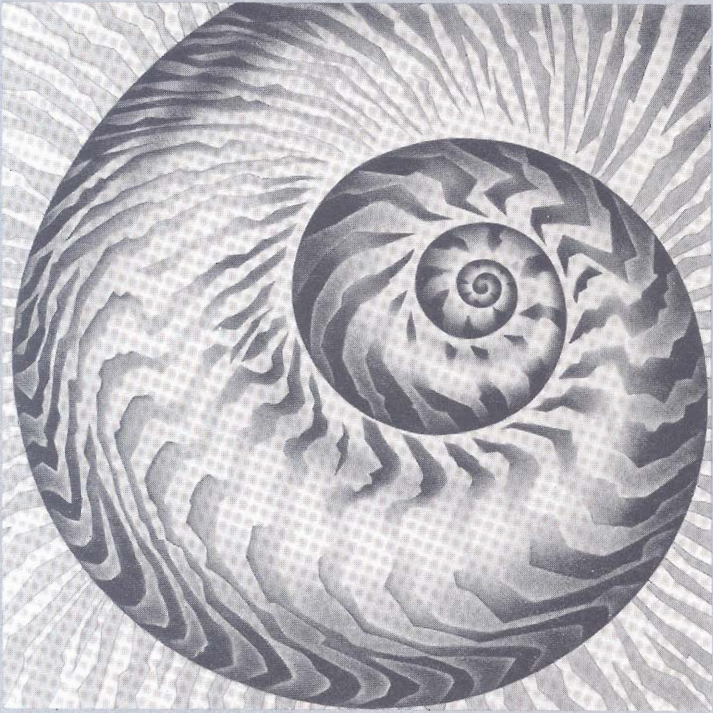
sinclair

QL·Macro Assembler



Software by

GST Computer Systems

QL·Macro Assembler

incorporating

QL·Linker and

QL·Screen Editor

First published in 1985
Sinclair Research Ltd
25 Willis Road
CB1 2AQ England

ISBN 1 85016 047 3

Copyright Notice

Macro Assembler and Linker Program and Documentation © GST Computer Systems Ltd 1985
Screen Editor Program and Documentation © Metacomco Ltd 1984
Packaging and Design © Sinclair Research 1985
Illustration © Jenny Tylden-Wright 1984.

Important Notes for New Users

The instruction manual for QL Macro Assembler is in three parts, describing in detail how to use each of the three programs which make up the package:

- Macro Assembler
- Linker
- Screen Editor

Each program's documentation is self-contained, but we recommend that you read the introductions to each section to familiarise yourself with their contents before you start.

Back-up copies

You are advised to make a back-up copy of each master cartridge using the clone program provided. Place the blank cartridge in Microdrive 1 and the master cartridge in Microdrive 2. Then type:

```
lrun mdv2_clone
```

and follow the instructions on the screen.

QL·Macro Assembler

Contents

1. Introduction

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

It tells you:

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

It does not:

- 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
- tell you how to talk to Qdos, the QL's operating system, for which you will have to consult the QL Technical Guide
- teach programming in general
- teach assembler programming or 68000 programming in particular.

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

1.1 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, %000000000000.

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> = ... | << | ...

At some points in the description of the macro facilities the characters [,], {, } must actually be coded as part of the assembler source program. Where it is not obvious whether these characters must be coded (in which case they are 'literal') or whether they are used as defined above to describe syntax (in which case they are 'metasymbols') their actual meaning is stated explicitly in each case.

2. How to run the assembler

You can Load and run QL Macro Assembler in one of two ways:

- **Interactive mode**

  In this mode the assembler will identify itself and prompt you for a command line. Upon completion of an assembly the assembler will prompt you for a further command line, so that you may perform several assemblies without reloading the program.

  When you have done all the assemblies you want you can terminate the assembler by replying to its prompt with a blank command line.

  You may run the macro assembler in interactive mode by any of the following commands where DEV is the device from which it is to be loaded (which may be any storage medium).

  – To run in parallel with the SuperBASIC interpreter:

  ```
      EXEC  DEV_MAC
  or: EX  DEV_MAC
  ```

  – To wait for completion of the assembler:

  ```
      EXEC_W  DEV_MAC
  or: EW  DEV_MAC
  ```

- **Non-interactive mode**

  In this mode the assembler receives its command line directly from the SuperBASIC interpreter and does not interact with you. On completion of the assembly the assembler will exit and will need to be reloaded if you wish to perform another assembly.

  You may run the macro assembler in non-interactive mode by one of the following commands:

- To run in parallel with the SuperBASIC interpreter:

```
EX  DEV_MAC; "<command line>"
```

- To wait for completion of the assembler:

```
EW  DEV_MAC;"<command line>"
```

where <command line> is described below. The quotes round the command line are required by the SuperBASIC interpreter.

Notes

The EX and EW commands are only available in the QL Toolkit and are not part of standard SuperBASIC.

The EX and EW commands allow you to pass data files to the program by specifying them after the program name. If any files are specified in this way they will be ignored by the assembler. See the QL Toolkit documentation for information on the full use of the EX and EW commands.

If you wish to change the screen window used by the macro assembler you may do so by running the program WINDOW_MGR and answering the questions it asks.

2.1 The command line

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 |
           –NOLINK
```

(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

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

–NOLINK normally the QL Macro Assembler generates output in S–ROFF format which must then be linked by the QL Linker; the –NOLINK option instructs the assembler to generate an output program file which can be run directly

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 an –ERRORS or –LIST option, or may be allowed to default. If no <listing> <file name> is given in an –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 _REL in the normal case or _BIN if –NOLINK has been coded.

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_REL

```
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_REL
```

Assemble MDV1_FRED_ASM, send an error listing only with no symbol table to MDV1_FRED_LIST, and put the binary in MDV2_FRED_REL (note that coding MDV2_FRED would not have achieved this)

```
MDV1_FRED SER1 MDV2_FRED_REL-ERRORS-SYM
```

assemble MDV1_FRED_ASM, print an error listing plus symbol table directly and put the binary in MDV2_FRED_REL

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 to be assembled:
*  this is rather
*  large so it is split into two separate
*  files
*
   INCLUDE MDV2_PROG1_IN
   INCLUDE MDV2_PROG2_IN

   END
```

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

It is recommended that file names of INCLUDEd files end in _IN, but this is not essential, and you can call them anything you like.

3.3 Screen output

The assembler writes a certain amount of information to the screen to let the user know what is happening. This includes a 'start' 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 30k) and the assembler's initial data space (about 4k).

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 an unexpanded QL it is possible to assemble a source file which occupies virtually the whole of a Microdrive cartridge as long as no other major task is running at the same time.

If you do several assemblies in one go (without reloading the assembler) then the assembler will return any memory it has obtained to the operating system at the end of each 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 is finished completely.

3.4 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 file name 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 file names 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.

3.5 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.

3.6 Object code output

3.6.1 Relocatable (S–ROFF) output

Normally the assembler will produce a relocatable binary output file in S–ROFF format (the standard Sinclair relocatable output file format). This output file may be linked using the Sinclair QL Linker with other files in the same format generated by the QL Macro Assembler and/or files in the same format generated by compilers for other languages.

Each assembly generates a single **module** (see the QL Linker section for more information about the details of the S–ROFF format and how to link together S–ROFF object files).

If you code a MODULE directive somewhere in your source program then that directive will specify the name of the module. If you do not code a MODULE directive then the assembler will construct one from the name of the primary source file by stripping the _ASM off the end of the file and stripping the first component (assumed to be a device name such as MDV2_) off the beginning.

For example, if the primary input file is called:

FLP2_SYSTEMX_PART3_ASM

then the default module name will be:

SYSTEMX_PART3

Other information is included as part of the module directive in the S–ROFF file, including the name of the assembler and the time and date of the assembly.

3.6.2 Directly executable output

Alternatively the QL Macro Assembler may generate a directly executable output file which may be run as a program using the EXEC or EXEC_W command without any need for linking. To make use of this option you must code –NOLINK In the command line (see section 2 above) and you must not use most of the assembler directives which relate to linker functions. See the description of each directive for full details.

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 allowed to overflow onto the following line, and these overflows will be taken into account when determining whether a page is full.

The listing output is paginated with the total page length defined by the user in a PAGELEN directive (q.v.) or allowed to default. To obtain essentially unpaginated output you may set PAGELEN 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:

<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

<blank>	is a blank line (i.e. a line feed character)
<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	macro flag	blank or +
6–7	section number	2-digit hex
8	(blank)	
9–16	location counter	8-digit hex
17	(blank)	
18–29	generated code	up to 12 digits hex

30	(blank)	
31–132	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 or macro from which it came and regardless of whether the line is listed or not.

The macro flag is blank if the line being listed came directly from an input file or contains the character '+' if the line was generated by a macro.

The section number is an internal number used to indicate which SECTION is being assembled; this number ties up with the section number given in the list of sections in the symbol table listing. It is left blank when absolute addresses (such as those generated under the influence of an OFFSET or ORG 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 to 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.

The source line printed on the listing is normally the fully expanded version of the line after values have been substituted for all macro parameters, functions and variables. However in the case of an error occurring during substitution, a partially expanded form of the line may be listed with an error message giving the reason for the problem.

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, NOLIST, EXPAND or NOEXPAND 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 consists of three separate listings: a list of all the sections used in the assembly, the main cross-reference listing of normal user symbols, and a cross-reference listing of macros.

4.2.1 The section report

The section report precedes the main symbol table listing. It has one line for each section name or common block name used in the assembly, each line having the following form:

Columns	Field contents	Format
1–8	symbol	up to 8 characters
9	(blank)	
10–13	symbol type	SECT or COMM
14	(blank)	
15–16	section number	2-digit hex
17	(blank)	
18–25	size	8-digit hex

The size field contains the size of the section or common block.

As all sections are the same if the –NOLINK option has been selected and no common blocks can exist at all, this report does not appear when –NOLINK has been selected.

4.2.2 User symbol cross-reference

This report lists all user defined operand-type symbols and gives the line number for each occasion on which the symbol was used.

Columns	Field contents	Format
1–8	symbol	up to 8 characters
9	(blank)	
10–13	symbol type	see below
14	(blank)	
15–16	section number	see below
17	(blank)	
18–25	value	8-digit hex
26	(blank)	
27-PAGEWID	cross-references	see below

The symbol type field contains one of:

MULT	the symbol is multiply defined; the assembler will use the first definition and print error messages for subsequent ones
XREF	the symbol is defined by an XREF directive
XDEF	the symbol is used in an XDEF directive
REG	the symbol is a register list defined by a REG directive
blank	anything else

The section number field only contains useful information if the symbol type field is blank (or XDEF) in which case it is one of:

blank	symbol depends on no section or common block base addresses
number	symbol depends (with a count of +1) on one common block or section base address, and this is the relevant section number
XX	symbol depends on more than one section or common block base address or depends on one but with a count other than +1

In the –NOLINK case absolute symbols will have this field blank and relocatable symbols will have the number 00 printed.

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

The rest of the line (up to the defined PAGEWID) will be filled with cross-reference information. If there is more than enough of this to fill the line it will continue on subsequent lines starting at column 27.

Each cross-reference consists of six characters as follows:

1–4	line number	4-digit decimal
5	definition flag	blank or '*'
6	(blank)	

and gives the number of a line on which the symbol was used. If the use of the symbol is a defining occurrence then the line number is followed by an asterisk.

Cross-references for a particular symbol are printed in ascending order of line numbers.

4.2.3 Macro cross-reference

This is a cross-reference listing of all macros involved in the assembly. It is in the same forrnat as the symbol cross-reference listing but the symbol type, section number and value fields are all blank.

A. Bibliography

QL Technical 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 machine code programs for the QL. It does not attempt to teach 68000 programming.

Available from Sinclair Research Limited, Stanhope Road, Camberley, for £14.95 mail-order.

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 from GST Computer Systems Limited, 91 High Street, Longstanton, Cambridge, for £8.95 mail-order.

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 from GST Computer Systems Limited, 91 High Street, Longstanton, Cambridge, for £8.95 mail-order.

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.

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. There are some exceptions to this which are concerned with the macro facilities of the assembler.

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.

A semicolon (;) at any position in a line (as long as it is not inside a <string> or an <arbitrary string>) introduces a comment; the semicolon and the rest of the line are ignored.

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 three types of <symbol> used by the assembler. <symbol>s appearing in the operation field are 'operation type symbols', those appearing in most operand fields are 'operand type symbols' and those appearing in the operand of a SECTION or COMMON directive are 'section names'. These 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 various 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

There is a special form of string used in some macro and conditional assembly directives:

<arbitrary string> = any sequence of characters not including space or comma ',' or backslash '\' or semicolon ';' |

{any sequence of characters}

where the {} are literal (i.e. they must be coded and are not part of the syntax description).

There is a special set of operators used in the conditional assembly directives:

<compop> =< | <= | >= | > | ˜= | <>

where ˜= and <> are alternate ways of coding "not equals"

Note that in macro calls and some conditional assembly directives the backslash character '\' is used to indicate that the statement is continued on the following line of input.

Note that the open square bracket character '[' is used to indicate variable substitution and may not appear in any other context (e.g. it may not appear with any other meaning in a <string> or <arbitrary string> or comment).

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.

Some macro and conditional assembly directives may be coded over more than one source line; any such line which is to be continued on the next line ends with a backslash '\' (optionally followed by comment). Full details are given when the directives concerned are described.

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
– a macro name

B.2.3 The operand field

The syntax of the operand field depends on the operation. <white space> terminates the operand except in the case of a macro call or a conditional assembly directive.

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.

It is, however, good practice to use the semicolon (;) to introduce comments on macro calls and conditional assembly directives as this will avoid confusing both the assembler and the human reader.

B.3 Expressions

Expressions are constructed from:

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

<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:

1 parenthesised expressions are evaluated first (in the natural way)
2 operators are evaluated according to priority; the order of priority is (highest first):

   unary +, -
   <<, >>
   &, !
   *, /
   binary +, -

3 operators of the same precedence at the same nesting level of parentheses are evaluated from left to right.

B.3.1 Values

A value (of a symbol or of an <expr> or of a partially evaluated sub-expression etc.) consists of a numeric term (4 bytes) and a list of relocation bases to be added or subtracted.

See B.3.2. below for details of which symbols have which values.

Values can be classified into various types by the following properties:

- **Addressing mode**
  This is an indication of the requested addressing mode required and is one of:

  normal — no specific request; interpret it as absolute or relocatable depending on the relocation factor

  XREF.S — the value consists of a single symbol which was declared in an XREF.S directive

  XREF.L — the value is either a more complicated construction involving XREF.S symbols or contains a reference to a symbol declared in an XREF.L directive

- **Relocation factor**
  This is the number of times the value is expected to be relocated finally by both assembler and linker with respect to the start address of the whole program. Each XREF (but not XREF.S or XREF.L) and label defined within a SECTION added into the value contributes +1 to this count and each such symbol subtracted from the value contributes −1 to this count.

  If the relocation factor is 0 the value is regarded by the assembler as **absolute.**

  If the relocation factor is 1 the value is regarded by the assembler as **simple relocatable.**

  If the relocation factor is anything else the value is regarded by the assembler as **complex relocatable.**

- **Number of relocation bases**
  This is the number of different XREF[<xlen>] symbols and base addresses of SECTIONs involved in the value (after any cancelling out has been done).

- **COMMON dependency**
  This is an indication of whether any symbol forming the value was the name of a COMMON section.

B.3.2 Values of various operand types

This section lists the various operands and describes the type of value they possess.

- **Numbers and strings**
  Numbers and strings have a value whose numeric term is the value of the number or string.

Addressing mode:	normal
Relocation factor:	0
Relocation bases:	none
COMMON dependency:	no

- **The current location counter**
  The value of the current location counter (*) is equal to the value a label coded on the same line would have, and the value is of identical form.

- **Labels**
  Symbols which are defined as labels in range of OFFSET, ORG or COMMON directives have values whose numeric term is the numeric value of the symbol.

Addressing mode:	normal
Relocation factor:	0
Relocation bases:	none
COMMON dependency:	no

Symbols which are defined as labels in range of SECTION directives have values whose numeric term is the offset of the label from the start of the section (within the module).

Addressing mode:	normal
Relocation factor:	+1
Relocation bases:	1: start address of section
COMMON dependency:	no

- **Symbols defined in XREF directives**
  Symbols which are defined in (any type of) XREF directives have a numeric term of zero and a single relocation base which is the external reference to the symbol.

  For symbols defined by XREF:

Addressing mode:	normal
Relocation factor:	+1
Relocation bases:	1: the symbol
COMMON dependency:	no

  For symbols defined by XREF.S or XREF.L:

Addressing mode:	XREF.S or XREF.L
Relocation factor:	0 (but irrelevant to the user)
Relocation bases:	1: the symbol
COMMON dependency:	no

- **Section names**
  Section names as used in SECTION directives are not operand type symbols, cannot be referred to anywhere other than in SECTION directives, and have no value

- **Common block names**
  Common block names have values whose numeric term is zero.

Addressing mode:	XREF.L
Relocation factor:	0 (but irrelevant to the user)
Relocation bases:	1: start address of the common block
COMMON dependency:	yes

- **Symbols defined by EQU**
  The value of a symbol defined by an EQU directive is the value of the <expr> coded on the EQU directive. For a definition of how values of expressions are derived, see below.

- **Undefined symbols**
  Symbols which are undefined at the point of reference (usually because they are forward references but sometimes because they are errors) are treated as labels defined in range of a SECTION directive.

B.3.3 Rules for operator processing

This section describes how the various operators combine values to make new values. See above for details of the actual arithmetic operations performed.

- **Unary +**
  This operator is ignored.

- **Unary –**
  The subexpression:

  ```
  -<subexpr>
  ```

  is treated in identical fashion to:

  ```
  (0-<subexpr>)
  ```

  (taking due account of operator priorities), see the description of binary subtraction below.

- **Binary addition**
  Addition of two normal operands will result in a normal value.

  The relocation factor of the result will be the sum of the relocation factors of the operands.

  The relocation bases involved in both operands are added together. If a particular relocation base occurs with a positive sign in one operand and a negative sign in the other it is cancelled out.

  Addition of two operands at least one of which is of type XREF.S or XREF.L will result in a value of type XREF.L. The relocation factor and relocation bases are kept track of in the same way as for the normal case.

- **Binary subtraction**
  Subtraction of two normal operands will result in a normal value.

  The relocation factor of the result will be the difference of the relocation factors of the operands.

The relocation bases involved in both operands are subtracted in the appropriate direction. If a particular relocation base occurs with the same sign in both operands it is cancelled out.

Subtraction of two operands at least one of which is of type XREF.S or XREF.L will result in a value of type XREF.L. The relocation factor and relocation bases are kept track of in the same way as for the normal case.

- **All other operators**
  These operators are only valid if both operands are of the following form:

Addressing mode:	normal
Relocation factor:	0
Relocation bases:	none
COMMON dependency:	no

  and will produce error messages otherwise.

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

<ireg>          = <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 following addressing modes are called (by Motorola) 'effective address' and can be coded (or at least a subset of them) in any instruction which has a general effective address as an operand:

```
<ea> = <dreg> |                D register direct
       <areg> |                A register direct
       (<areg>) |              register indirect
       (<areg>)+ |             postincrement
       -(<areg>) |             predecrement
```

<table>
<tr><td><expr>(<areg>) |</td><td>indirect with displacement</td></tr>
<tr><td><expr>(<areg>,<ireg>) |</td><td>indirect with index</td></tr>
<tr><td><expr> |</td><td>absolute short</td></tr>
<tr><td><expr> |</td><td>absolute long</td></tr>
<tr><td><expr> |</td><td>PC relative</td></tr>
<tr><td><expr>(PC) |</td><td>PC relative</td></tr>
<tr><td><expr>(PC,<ireg>) |</td><td>PC with index</td></tr>
<tr><td>#<expr></td><td>immediate</td></tr>
</table>

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>, for example, 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.

The addressing mode generated depends on whether the referring instruction is in absolute code (in the range of an ORG) or relocatable code (in the range of a SECTION). This table summarises the generated addressing modes:

From	**To**	**Generates**
abs	abs	absolute short or long as appropriate
	reloc	absolute long
	forward	absolute long
	XREF	absolute long
	XREF.S	absolute short
	XREF.L	absolute long
reloc	abs	absolute short or long as appropriate
	reloc	PC-relative
	forward	PC-relative
	XREF	PC-relative
	XREF.S	absolute short
	XREF.L	absolute long

If the value of the expression is complex relocatable the assembler will produce an error message.

Forward references within absolute code will always be generated as absolute long addresses. You can code an explicit (PC) to make such references PC-relative, but there is no way to force them to be absolute short.

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
```

(within a SECTION) is not legal and will generate an error, whereas:

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

(within a SECTION) 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 not very far away
```

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 not 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. 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.

A dot '.' as the last character of an instruction (or directive or macro) name in the operation field of a source line is ignored (e.g. the exchange instruction may be coded as EXG, EXG. or EXG.L). This feature is sometimes useful when designing macros.

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

MOVE.L	an instruction with an operand length coded
BEQ.S	an instruction with an extent coded
JSR	an instruction with no extra bits

```
MOVE.L   D0,A0      automatically generates MOVEA.L

MOVE.L   #2,D3      automatically generates MOVEQ.L
```

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[<length>]    <dreg>,<ea>
BCHG[<length>]    #<expr>,<ea>

BCLR[<length>]    <dreg>,<ea>
BCLR[<length>]    #<expr>,<ea>

BSET[<length>]    <dreg>,<ea>
BSET[<length>]    #<expr>,<ea>

BTST[<length>]    <dreg>,<ea>
BTST[<length>]    #<expr>,<ea>
```

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:

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

<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 Scc instruction has the same set of conditions as DBcc but not the RA synomym:

```
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
ORG	absolute program section
COMMON	COMMON section
RORG	adjust current location
OFFSET	define offset symbols
DATA	specify data space
END	end of program
XREF	refer to external symbols
XDEF	define symbols to be external
MODULE	define module name for the linker
COMMENT	include comment in linker listing

The following directives require labels:

EQU	assign value to symbol
REG	define 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
PAGELEN	define length of page
LIST	switch listing on

NOLIST	switch listing off
TITLE	define title for listing

There are a number of other directives which are involved in the macro and conditional assembly facilities and these are described below in section B.7.

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 at least 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

The directive:

SECTION <symbol>

specifies that following instructions and data are to be placed in the named relocatable section. You may choose any names you wish for sections; these names may be the same as operand type symbols or operator type symbols.

If you have coded the –NOLINK option then all generated code is placed in one section and the <symbol>s given on SECTION directives are ignored.

The assembler insists that all instructions are coded within a section. Almost all programs must therefore contain at least one SECTION directive.

The start of a section within a module is forced (by the linker) to begin on an even address, but changing between sections within a module does not cause any automatic even address alignment.

For example:

```
SECTION   ONE
DC.B      1          this starts on an even address

SECTION   TWO
.......   ...        (anything)

SECTION   ONE
DC.B      2          this byte is at an odd address
```

The section names are neither operand type symbols nor operator type symbols and therefore you cannot refer to a section name from anywhere other than a SECTION directive.

It is however possible to have an operand type symbol with the same name as a section name, and it is possible to declare this name to be an external symbol, for example:

```
          SECTION   FRED

FRED:
          XDEF      FRED
```

In this example the symbol FRED refers to the first address in the subsection of FRED which resides in the current module, and this symbol is available to other modules which may refer to it using XREF.

B.6.3 ORG – start absolute section

The directive:

```
ORG <expr>
```

instructs the assembler to generate code at the absolute address specified by <expr>, which must be absolute and contain no external or forward references.

The ORG directive is not permitted if the –NOLINK option has been coded.

The use of the ORG directive renders the resulting program position-dependent so that it will not normally be possible to run it as a Qdos program.

B.6.4 COMMON – start COMMON section

The directive:

COMMON <symbol>

introduces a common section in the same way as the SECTION directive introduces an ordinary section.

The COMMON directive is not permitted if the –NOLINK option has been coded.

This directive exists to allow declaration of and access to Fortran-style common blocks. This is not considered to be a generally useful feature and is included solely to enable assembler access to data structures used by Fortran (or other high level languages which make use of the Fortran COMMON scheme).

The COMMON directive creates the <symbol> of section type (as does the SECTION directive); it also creates an operand type symbol of the same name **as if it had been declared with an XREF.L directive.** The value of this symbol is an offset from a global origin of common blocks.

In the two cases which are not re-entrant (default and COMMON END – see the QL Linker manual) the global origin of common blocks is the start of the program, and in the re-entrant case (COMMON DUMMY) it is the start of the store area allocated to the common blocks (which is not known until run-time).

There is no way to tell the assembler which type of common allocation will be performed by the linker, but the way the assembler handles common allows most of the code to be the same for both cases.

The symbols declared as labels within a COMMON section have **absolute** values as if they were declared within range of an OFFSET 0 directive. They are intended to be used as offsets to an address register which holds the address of the base of the common block.

You must ensure that an address register is allocated throughout to hold the address of the base of all common blocks, and the initialisation of this register depends on whether code which is not re-entrant is being generated in which case something like:

```
LEA                     COMMBASE(PC),A5
```

will do, or whether re-entrant code is being generated in which case a call to the operating system to allocated memory space will return the address of the base of that space.

From here on the same code can be used in both cases: to obtain the base address of common block FRED above in A4:

```
MOVE.L      A5,A4       base of all common blocks
ADD.L       #FRED,A4    base of FRED
```

and you can then move data around in the common block in the same way in both cases, e.g.:

```
MOVE.L                  VAR1(A4),VAR2(A4)
```

Note that this code knows that VAR1 and VAR2 are within 32k bytes of the start of FRED but makes no assumptions about where the linker will put FRED in relation to the start of all common blocks. If you know that the whole program (in the non-re-entrant case) or the total size of all common blocks (in the re-entrant case) is less than 32k you may use:

```
ADD.W                   #FRED,A4
```

instead, and the linker will complain if there is any overflow.

Note that the use of the symbols VAR1 etc. in any other way is likely to be unhelpful, for example code like:

```
MOVE.L                  VAR1,VAR2
```

will do silly things like trying to copy parts of the operating system ROM around.

Within range of a COMMON directive DS and RORG directives may always be coded. If the code generated is to be non-re-entrant then DC and DCB directives may also be coded. In no circumstances may instructions be coded.

If re-entrant code is required (linker option COMMON DUMMY) and DC or DCB directives are coded within range of a COMMON directive then the linker will generate error messages.

B.6.5 RORG – adjust relocatable origin

The directive

RORG <expr>

resets the current location counter to <expr> bytes from the start of the current section or common section. RORG directives must only be coded following a SECTION or COMMON (with no intervening OFFSET or ORG).

If the <expr> in a RORG directive has a value higher than the address of any code generated in the section then the length of the section is increased to this value which will be used by the linker in performing address allocation.

The <expr> may be absolute or relocatable; in the latter case it must be simple relocatable with respect to the current section. It must contain no forward or external references and must not be negative.

The <expr> must not contain any symbols which are COMMON section names.

B.6.6 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.6.7 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

B.6.8 XREF – refer to external symbol

The directive:

```
XREF[<xlen>]            <symbol>{,<symbol>}
```

declares the listed <symbol>s to be external. Code within the current module may make references to these symbols and the references will be resolved by the linker.

The XREF directive for a symbol must occur before any other reference to the symbol otherwise the assembler will report an error. The XREF directive is not permitted if the –NOLINK option has been coded.

When a <symbol> declared in an XREF directive forms (possibly part of) an <expr> which is coded where a general effective address (<ea>) is required the user must give the assembler some help in choosing the addressing mode required. This is done via the <xlen> field.

<xlen> may be blank, in which case the assembler will usually generate PC-relative addressing modes when the <symbol>s are referred to, or '.S' in which case the assembler will generate absolute short addressing modes when the <symbol>s are referred to, or '.L' in which case the assembler will generate absolute long addressing modes when the symbols are referred to.

Note that (for example) a relocatable symbol may be referred to by XREF.L, in which case absolute long address references to it may be made. In most circumstances the linker will end up with the right answer for this sort of thing as long as you didn't want the program to be position independent.

The rules for the addressing mode chosen when an <expr> contains several symbols of different types are discussed at B.4.2 above.

A symbol may be declared in rnore than one XREF directive (so that, in particular, XREFs can be used in macros with no worries about duplicate declarations). (If the same symbol is defined more than once in the same XREF then some harmless error messages will result.)

B.6.9 XDEF – declare external symbol

The directive:

```
XDEF                          <symbol>{,<symbol>}
```

declares the <symbol>s to be external. These symbols should be defined in the current module and are made available to other modules by this declaration.

There are no positioning requirements on the XDEF directive, which may occur either before or after any other uses of the <symbol>s. The XDEF directive is not permitted if the –NOLINK option has been coded. A symbol may appear in more than one XDEF directive.

Due to restrictions imposed by the relocatable binary format and the linker only some types of <symbol> can be coded in an XDEF directive. The following types of <symbol> can be coded in an XDEF directive:

some absolute symbols, being:

labels following an ORG directive

symbols defined by an EQU directive which involve no (residual) external references and no (residual) SECTION or COMMON relocation bases

some symbols whose value is an offset from the start of a SECTION present in the current module, being:

labels following a SECTION directive

symbols defined by an EQU directive which involve no (residual) external references and only one (residual) SECTION relocation base with a relocation factor of 1.

The following types of symbol cannot be coded in an XDEF directive due to the relocatable binary and linker restrictions:

symbols defined by an EQU directive which involve one or more residual external references and/or more than one residual SECTION relocation base and/or any SECTION relocation base with a relocation factor other than 1.

B.6.10 MODULE – declare module name

The directive:

```
MODULE                  <title string>
```

is optional and specifies the contents of the source directive in the output relocatable binary file. Normally it is not necessary to code a MODULE directive and a default will be constructed by the assembler as described at 2.7.1 above.

The MODULE directive is ignored if the –NOLINK option has been coded.

B.6.11 COMMENT – include comment in binary

The directive:

```
COMMENT                 <title string>
```

places the string in the relocatable binary output file as a comment directive. This has no effect on anything except that it is included in the listing generated by the QL Linker.

The COMMENT directive is ignored if the –NOLINK option has been coded.

B.6.12 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 <expr> may include references to external symbols (defined by earlier XREF directives).

The value of the defined symbol is calculated as explained in B.3.1.

B.6.13 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.6.14 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>
```

This directive may be coded within the range of a SECTION or ORG directive. It may also be coded within the range of a COMMON directive but this should only be done if the code is to be non-re-entrant.

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 the 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. Thus 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 may contain any legal combination of external references.

Data to be generated may have any value type. However any data with a relocation factor other than zero will cause a warning message to be produced as the result is probably a program which is not position independent.

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.6.15 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 or external 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.6.16 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.

This directive may be coded within the range of a COMMON directive but this should only be done if the code is to be non-re-entrant.

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 or external 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> may contain any legal combination of external references.

Data to be generated may have any value type. However any data with a relocation factor other than zero will cause a warning message to be produced as the result is probably a program which is not position independent.

B.6.17 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.6.18 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 or external references and must be between 72 and 132 inclusive. If no PAGEWID directive is present the default is 132 characters.

B.6.19 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 or external references. The value given is the physical length of the paper; rather fewer lines of assembler source are actually listed on each page.

B.6.20 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.6.21 NOLIST – switch listing off

The directive

```
NOLIST
```

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

B.6.22 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.6.23 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 or external 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.

B.7 Macro facilities

A **macro** is a set of assembler source statements (both instructions and directives) which are given a name.

This set of statements may be included in your program at any point by coding the name of the macro in the operation field of a source line as if it were a user-defined instruction or directive.

Thus a macro can be used as a shorthand way of writing a set of statements which has to be repeated several times in a program.

A set of text substitution and conditional assembly facilities is also provided so that a macro need not generate exactly the same sequence of code each time it is used but can generate different code depending on parameters passed to it.

As all these facilities are interdependent it is not possible to arrange this section of the manual so as to avoid forward references entirely: you may find it necessary to read right through section B.7 quickly so as to gain a general understanding and then read it again to study the details.

B.7.1 Text substitution

The input to the assembly process may come from the input files or from a macro which is being expanded. In either of these cases (but not while a macro definition is being processed **or on a comment line introduced by an asterisk '*' in column 1)** any occurrence of:

[<variable>]

(where the [ ] are literal) will be replaced by the string value currently associated with the <variable>. There are two (syntactic) types of <variable>, being <symbols>s or calls to functions:

```
<variable>        =  <symbol>|
                     <function>

<function>        = <symbol>[(<parm>{,<parm>})]
```

```
<parm>                 =  <variable>|
                          <expr>|
                          <ea>
```

(where the [ ] and { } are metasymbols). A <parm> may be interpreted as a <variable> or an <expr> or an <ea> depending on the type of data required for that parameter by the particular function.

Substitutions can be nested. For example, suppose the current value of the variable N is the string '3' (not including the quotes) and the current value of the variable VAR3 is the string 'A value' (not including the quotes) then the assembler source:

```
DC.B                 '[VAR[N]]'
```

will be expanded to:

```
DC.B                 'A value'
```

For a list of functions provided in the assembler see B.7.4.

Note that the character '[' will always attempt to cause a substitution and so this is effectively a reserved character in that any attempt to code it as part of a string or comment will always cause either a substitution or an error message.

For a discussion of the scope of variables and macro parameters see B.7.3.

B.7.2 Macro definitions

A macro definition begins with the directive:

```
<label>      MACRO    [<symbol>{,<symbol>}]
```

the commas are optional and macro parameters can alternatively or in addition be separated by <white space>; in addition the comma or <white space> can be followed by a backslash character '\' in which case the rest of the line is ignored and the next parameter is taken from the next input line (where the [ ]{ } are metasymbols) and ends with the directive:

```
ENDM
```

Macro definitions may not be nested.

The <label> is defined to be the name of the macro. It is defined as an operator type symbol (and must not therefore clash with any directive, instruction or other macro name).

The list of <symbol>s gives names to the parameters to the macro. The macro parameters may then be accessed by substituting for the parameters as variables, e.g.:

```
FRED        MACRO     P1,P2
            ...
            DS.W      [P2]
            ...
            ENDM
```

Alternatively the functions .NPARMS and .PARM may be used to access the parameters and in this case no prior knowledge of the number of parameters to be coded is needed. Example:

```
FRED        MACRO     P1,P2
            ...
            DS.W      [.PARM(2)]
            ...
            ENDM
```

has the same effect as the previous example.

When a macro definition is encountered in the assembler's input it is scanned without any string substitution until an ENDM directive is found. The ENDM must therefore be coded directly and must not be generated by substitution! This also applies to the MACLAB directive (see B.7.7).

B.7.3 Defining variables

Variables may be declared as local to a macro either by their appearance as macro parameters on the MACRO directive or by their appearance in LOCAL directives. This does not assign any values to the variables.

Variables are given values by SETSTR or SETNUM directives. In addition variables which are macro parameters are given values when the macro is called.

The values of variables are used when the variable name appears in square brackets[ ].

■ The LOCAL directive

The directive

```
LOCAL      <symbol>{,<symbol>}
```

(where the { } are metasymbols) declares the <symbol>s to be local variables within the macro currently being expanded. None of the <symbol>s may appear as a parameter name for the current macro or in another LOCAL directive in the same macro.

For a full discussion of the scope of variables see below.

■ The SETSTR directive

The directive

```
<label>      SETSTR      <arbitrary string>
```

defines the <label> to be a variable and assigns it the <arbitrary string> as its value.

See below for a discussion of the scope of variables. Variables can be set (using SETSTR and/or SETNUM, see below) as often as you like during an assembly.

When setting a variable to a string value which does not contain any special characters which might confuse the assembler (space, comma, semicolon) it is not necessary to code the curly brackets { } round the <arbitrary string>, for example:

```
FRED        SETSTR      The.value.of.bert
BERT        SETSTR      197.999
WOMBAT      SETSTR      [FRED]=[BERT]
```

When one of these special characters is required the curly brackets must be coded, for example:

```
STRING      SETSTR      {This has spaces in it}
STRING2     SETSTR      {'and so does this'}
STRING3     SETSTR      {don't take; semicolon as comment}
NULL        SETSTR      {}          how to set a null string
DANGER      SETSTR      {[A]} don't know if A contains spaces
```

■ The SETNUM directive

The directive

```
<label>     SETNUM      <expr>
```

evaluates the <expr>, converts the final numeric result to a string, and assigns this string to the variable <label> in the same way as for SETSTR.

The use of SETNUM is often for counting;

```
COUNT       SETNUM      [COUNT]+1
```

in conjunction with conditional assembly for the generation of tables etc.

The following example shows the difference between SETNUM and SETSTR:

```
STR         SETSTR      1+1+1
NUM         SETNUM      1+1+1

;       STR = [STR]       (must use ';' because '*' comments
;       NUM = [NUM]       are not expanded)
```

The above two lines of comment will be expanded to:

```
;      STR = 1+1+1      (must use ';' because '*' comments
;      NUM = 3          are not expanded)
```

■ Scope of variables and macro parameters

Variables and functions occupy a separate name space from all others; there is in particular no danger of name clashes with ordinary labels. Within this name space the names of functions are unique. The names of macro parameters and LOCAL variables are unique within a macro but may be duplicated in other macros or by global variables.

A macro parameter is in scope for the duration of the expansion of that macro. It is effectively created, and given the appropriate value, at macro call time, and deleted when the macro expansion has finished.

Macro parameters may be set to new values by SETSTR or SETNUM directives.

Variables which appear in LOCAL directives are similarly in scope for the duration of the expansion of the macro in which the LOCAL directive appears (but only from the LOCAL directive to the ENDM directive: any use of the same name before the LOCAL directive is processed refers to the global variable of the same name).

The scope of a variable which does not appear in a relevant LOCAL directive is global: at any point after the first definition of the symbol it may be used in substitution, regardless of the macro generation levels of both the definition and the substitution.

Functions are global except that some cannot usefully be called outside a macro. Such calls will produce null strings or error messages depending on the individual function.

When a variable to be substituted is encountered, the macro parameter or LOCAL variable within the current macro of that name is used, if any. Otherwise the global variable of the same name is used. If there isn't one of these either then an error message is generated.

B.7.4 Functions

The assembler contains a number of functions which can be used in string substitutions. The names of these functions are all <symbol>s which begin with a dot '.' to help avoid confusion with ordinary user variables. It is not possible for the user to define his own functions.

For example:

```
DS.B        [.LEN(VAR3)]
```

will expand (with the value of VAR3 given earlier) to:

```
DS.B        7
```

and:

```
DC.B        '[.LEFT(VAR[N],[.LEN(VAR[N])] - 1)]'
```

will first expand to:

```
DC.B        '[.LEFT(VAR3,[.LEN(VAR3)]–1)]'
```

and will then expand to:

```
DC.B        '[.LEFT(VAR3,7-1)]'
```

At this point the .LEFT function will be expanded: as it requires a number for its second parameter an attempt is made to evaluate the second parameter as an ordinary <expr>. The result of this operation is:

```
DC.B        '[.LEFT(VAR3,6)]'
```

and the final substitution gives:

```
DC.B        'A valu'
```

The individual functions available are listed in paragraphs below.

■ .DEF – whether variable is defined

The function

.DEF(<symbol>)

returns the string 'TRUE' if the <symbol> has previously been defined by a SETNUM or SETSTR directive or the string 'FALSE' otherwise, in both cases not including the quotes.

■ .LEN – length of a string

The function

.LEN(<variable>)

returns (as a string, e.g. '7') the length in characters of the string represented by the <variable>.

■ .LEFT – left substring

The function

.LEFT(<variable>,<expr>)

returns a string consisting of the leftmost <expr> characters of <variable>. If <expr> is zero or negative the result is the null string. If <expr> is greater than the length of the string <variable> then the result is identical to <variable>.

■ .RIGHT – right substring

The function

.RIGHT(<variable>,<expr>)

returns a string consisting of the rightmost <expr> characters of <variable>. If <expr> is zero or negative the result is the null string. If <expr> is greater than the length of the string <variable> then the result is identical to <variable>.

■ .INSTR – locate substring

The function

.INSTR(<variable>,<variable>)

returns as a string the character position of the first occurrence of the second <variable> as a substring in the first <variable>. The first character in the first variable is regarded as character position 1. If there is no match then .INSTR returns zero.

■ .UCASE – convert to upper case

The function

.UCASE(<variable>)

returns as a string its parameter with all lower case letters converted to upper case. The use of this function to identify macro parameters is highly recommended as it means that the user of the macro does not need to code the parameters in any particular case.

■ .NPARMS – number of parameters

The function

.NPARMS

returns the number of parameters that have been passed to the current macro. If called outside any macro it is an error.

■ .PARM – macro parameter

The function

.PARM(<expr>)

returns the <expr>th parameter to the current macro. If called outside any macro, or if the macro had less than <expr> parameters on the current call, then an error is generated.

This error case can be avoided by checking against .NPARMS before using .PARM.

■ .LAB – macro call label

The function

```
.LAB
```

returns the label that was coded on the macro call. If no label was coded the null string is returned. If this function is coded outside a macro then an error is generated.

■ .EXT – macro call extension

The function

```
.EXT
```

returns the last character of the macro name coded if the penultimate character of the macro name was dot '.'. If the macro call had no extension then .EXT returns the null string. If this function is used outside a macro then an error is generated.

■ .L – unique label generation

The function

```
.L
```

will give a different four-digit number for each macro expansion in which it is used. This is useful for generating local labels that will not clash with other labels generated by others macros or other invocations of the same macro.

If this function is used outside a macro then an error is generated.

Example: suppose a label is needed inside some code generated by a macro, and the macro is likely to be called several times in the assembly:

```
LOOP[.L]
              .....
              .....
              DBRA          D3,LOOP[.L]
```

■ .OTYPE – type of operand

The function

.OTYPE(<ea>)

investigates the <ea> as if it were an operand to an instruction and, depending on the operand type, returns one of the following strings:

Result string	**Format of <ea>**
DREG	<dreg>
AREG	<areg>
IND	(<areg>)
INDDEC	-(<areg>)
INDINC	(<areg>)+
DISPL	<expr>(<areg>)
INDEX	<expr>(<areg>,<ireg>)
EXPR	<expr>
PC	<expr>(PC)
PCINDEX	<expr>(PC,<ireg>)
IMMED	#<expr>
MULT	register list
USP	USP
CCR	CCR
SR	SR
ERROR	anything else

■ .ABS – enquire type of value

The function

.ABS (<expr>)

returns TRUE if the <expr> has no relocation bases, i.e. the numeric part is all there is to say about it, or FALSE if some relocation bases are involved.

Together with .OTYPE above, this function enables macros to detect things like a parameter having value '#FRED' where FRED was an EQU symbol with value of absolute zero. This is likely to be helpful when trying to write macros to generate structure statements.

■ Assembler environment enquiries

The functions:

.TIME	of assembly
.DATE	of assembly
.FILE	primary source file being assembled
.VER	version number of the assembler
.OS	host operating system ('Qdos' or '68K/OS')

are provided so that the program can know something about how it is being assembled.

B.7.5 Listing control

■ Listing of substituted lines

Normally a line is fully expanded and then processed by the assembler as before. This means that (only) the fully expanded form is listed.

When the assembler processes macro definitions it does not expand substitutions, and does not act on most of the data contained in the macro body. Lines of macro definitions are printed as coded, without any expansion.

When an error is encountered during expansion (such as wrong brackets of various types'[ ]{}( ),', missing parameters to functions, failure to evaluate <expr>s as desired) the current partly-expanded state of the line is listed, with error messages as appropriate, and no further processing is applied to the line.

■ Listing of macro-generated lines

Macro definitions are listed or not in the normal way subject to the normal options and directives, except that any listing control directives present inside the macro definitions themselves are not acted on.

Macro calls are listed or not in the normal way subject to the normal options and directives, which may include macro expansion controls if the macro call was generated by a macro.

Lines generated by macros are listed subject to the –NOLIST, –ERRORS and –LIST options and the LIST and NOLIST directives in the usual way. In addition the directive NOEXPAND switches off listing of lines generated by macros until either the EXPAND or ENDM directive is met. If the EXPAND directive is met, listing is switched back on again. If the ENDM directive for the macro in which the NOEXPAND appeared is met, the listing status reverts to what it was in the calling macro. A NOEXPAND directive may be coded outside all macros, in which case no macro expansions are listed unless they contain EXPAND directives.

The NOEXPAND and EXPAND directives themselves are not listed.

No special form of comment which overrides expansion suppression is provided as this can be achieved with:

```
                EXPAND
;               Macro FRED has generated a [WOMBAT]
                NOEXPAND
```

Note that in order to debug a macro you may have to test it with all the NOEXPAND directives missing, and put these in later when you have got it basically working. You can of course write all your NOEXPAND directives as

```
[DEBUG]EXPAND
```

and SETSTR DEBUG to NO or {} as required.

Two directives are provided to generate error and warning messages respectively. These can be used by a macro which checks its parameters for validity to tell the programmer that wrong or suspect parameters have been coded.

```
ERROR
WARNING
```

The directive is listed, together with an error or warning message respectively, regardless of the state of NOEXPAND and NOLIST (as are any other lines in macro expansions which generate error messages). Normally comments would be included on these directives to tell the user what he has done wrong:

ERROR	Parameter P1 should not be [P1]
WARNING	Use of XPQ option would generate less code

B.7.6 Macro calls

A macro is called by coding:

[<label>] <symbol> [<arbitrary string>{,<arbitrary string>}]

(where the [ ] { } are metasymbols) where:

<label> is passed through to the body of the macro as the value of the function .LAB

<symbol> is the name of the macro, optionally with a one-character extension preceeded by a dot '.'; if a legal extension is coded this is passed through to the body of the macro as the value of the function .EXT

<arbitrary string>s are macro parameters

the commas are optional and macro parameters can alternatively, or in addition, be separated by <white space>; in addition the comma or <white space> can be followed by a backslash character '\' in which case the rest of the line is ignored and the next parameter is taken from the next input line.

Note that as parameters can be separated by spaces, everything on the line will be assumed to be macro parameters, including the comment, unless you do something about it. What you do is to precede any comment (on the last continuation line) with a semicolon ';'.

There is virtually nothing you can do wrong, as far as the assembler is concerned, when coding a macro call. If too few parameters are coded then the remainder of the named parameters in the macro will be assigned null string values. If too many parameters are coded this is not an error because you can access them within the macro using the .NPARMS and .PARM functions. If no label or extension is coded the functions .LAB and .EXT will return null strings.

Macro calls may be nested to any depth (subject only to running out of memory). Macro calls may be recursive.

Example:

```
MAKETAB  27,   49,   123, 99, \ first row of table
          1,   99,  0,    3,    \ second row of table
          5,   8,    187, -1    ; last row of table
```

B.7.7 Conditional assembly facilities
Conditional assembly is provided within macros only.

The facilities provided are:

- the ability to define a special sort of label

- a conditional GOTO directive which either does or does not resume expansion of the macro at a specified label depending on a string or numeric comparison of two values

■ **The MACLAB directive**
The directive:

<label> MACLAB

defines the label. This directive is processed during macro definition and must not contain any string substitutions (except, perhaps, in the comment).

The scope of the <label> is local within the macro in which it is defined. Jumping to a macro label leaves you at the same recursive level, if the macro has been called recursively.

■ **The IFSTR, IFNUM and GOTO directives**
The directive:

■ **IFSTR <arbitrary string>,<compop>, <arbitrary string>,GOTO,<label>**
where the commas may be preceded or followed or replaced by <white space>, and backslash may be used to introduce continuation lines, as for macro calls

means 'perform a string comparison, and if the condition is true resume assembly at the line containing a MACLAB definition of the <label>'. The GOTO is a noise word and can be omitted.

Similarly IFNUM makes a numeric comparison between two <expr>s:

- **IFNUM <expr>,<compop>,<expr>,GOTO,<label>**

Examples:

```
          IFSTR     [.LEFT(P2,1)] = D GOTO DREG
          IFSTR     {[P3]} = {} GOTO EXIT

COUNT     SETNUM    1

LOOP      MACLAB
          DC.L      [.PARM([COUNT])]
COUNT     SETNUM    [COUNT]+1
          IFNUM     [COUNT] <= [.NPARMS] GOTO LOOP
```

As a piece of syntactic sugar a GOTO directive is provided so that:

GOTO <label>

can be coded instead of garbage such as:

```
          IFSTR     {   } = {   } GOTO <label>
```

Note that if there is any chance of an <arbitrary string> expanding to something containing spaces or commas then the {   } must be coded. For example, if you code:

```
          IFSTR     [FRED] = 99 GOTO LABEL
```

and FRED has the null string as value, this will expand to:

```
          IFSTR     = 99 GOTO LABEL
```

which is an error of some sort, whereas:

```
          IFSTR     {[FRED]} = 99 GOTO LABEL
```

will be much safer.

B.8 The macro library

Included with the QL Macro Assembler is a file containing definitions of some useful macros. This section describes those macros as supplied, but you may of course add to them and modify them if you need extra features.

B.8.1 Common features

The following features are common to all relevant macros.

■ **Length of jumps**

Some macros generate forward branches. These branches will be short unless .L is explicitly specified as part of the appropriate macro or parameter.

■ **Reserved Identifiers**

All names generated by macros which are not local to a macro start with a dot; this includes all variable names and any generated symbol names (including labels and register lists). To avoid clashes with the names in the library user variables and symbols should not start with a dot when you are using the macro library.

■ **THEN and DO**

Any macro which requires the word THEN as a parameter will accept DO as a synonym and vice versa.

B.8.2 Syntax definitions

The following syntax definitions are used in the descriptions of the macros.

■ **Simple condition**

```
<scond>      =   <relop>|
                 <ea> SET <ea> |
                 <ea> CLEAR <ea>
```

These combinations of simple tests allow the user to test condition codes or to perform a generalised compare or to test the value of a bit.

The SET and CLEAR comparisons test the state of a bit. The first <ea> a bit number in a form acceptable to a BTST instruction and the second <ea> is the address of the operand in which the bit resides (also in a form acceptable to a BTST instruction).

■ Condition

```
<cond>        =  <scond>|
                 <scond> OR <scond>|
                 <scond> AND <scond>
```

■ Relational operator

```
<relop> = <cc>[<length>]

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

The <length> on the <relop> is the size of the compare instruction generated to give the condition result.

B.8.3 IF and associated macros

The macros IF, ELSEIF, ELSE and ENDIF may be used to write code which tests values of operands and executes one of a number of pieces of code depending on those values. Use of these macros can avoid having to write a lot of error-prone CMP and Bcc instructions.

The general structure which you may construct using these macros is:

```
IF   <cond>   THEN[<extent>]
.
.
{
      ELSEIF[<extent>]   <cond>   THEN[<extent>]
        .
        .
}
[
        ELSE[<extent>]
        .
        .
]
ENDIF
```

■ The IF macro

The IF macro is allowed anywhere where code is allowed. The <extent> on the THEN parameter is the distance to the corresponding ELSEIF, ELSE or ENDIF macro.

- **The ELSEIF macro**

The ELSEIF macro is allowed only after a corresponding IF or ELSEIF macro.

The <extent> on the ELSEIF macro is the length of branch to the ENDIF macro (this branch is taken if the **previous** test was successful). The <extent> on the THEN parameter is the length of branch to the next ELSEIF, ELSE or ENDIF macro (this branch is taken if the test is not successful).

- **The ELSE macro**

The ELSE macro is only allowed after a corresponding IF or ELSEIF macro.

The <extent> on the ELSE macro is the length of the branch to the corresponding ENDIF macro.

- **The ENDIF macro**

The ENDIF macro is allowed after a corresponding IF, ELSEIF or ELSE macro.

B.8.4 FOR and associated macros

The macros FOR and ENDFOR allow you to code a loop which will execute a given number of times (possibly zero).

The structure of a FOR Loop is:

```
FOR [<length>]   <ea> = <ea>   <a>   [<b>]   DO[<extent>]
ENDFOR
```

where:

```
<a> = TO <ea> | DOWNTO <ea>
<b> = STEP <ea> | BY <ea>
```

- **The FOR macro**

Either the TO parameter must be coded, in which case you should ensure that the STEP is a positive number and the loop counts upwards, or the DOWNTO parameter must be coded, in which case you should ensure that the STEP is a negative number and the loop counts downwards.

The test for loop termination uses signed arithmetic in all cases.

The STEP parameter gives the amount to be added to the loop index each time round. If you omit it then '#1' is assumed for a TO loop or '#-1' is assumed for DOWNTO loop. The word BY may be used instead of STEP.

The <length> on the FOR macro is the length of all instructions generated for setting, stepping and comparing the counter. The <extent> on the DO parameter is the length of the branch to the ENDFOR macro.

■ The ENDFOR macro

The ENDFOR macro may only be used after a corresponding FOR macro.

B.8.5 Loop macros

The macros WHILE, ENDWHILE, REPEAT, UNTIL and FOREVER allow you to code loops that execute repeatedly while some condition is true or until some condition is true.

WHILE and ENDWHILE generate a loop that tests its condition at the beginning of the loop.

REPEAT and UNTIL generate a loop that tests its condition at the end of the loop; this loop is always executed at least once.

REPEAT and FOREVER generate a loop that executes forever; this is sometimes useful for the main loop in a program.

```
WHILE <cond> DO[extent>]
          .
          .
ENDWHILE

REPEAT
          .
          .
UNTIL <cond>

REPEAT
          .
          .
FOREVER
```

- **The WHILE macro**

The WHILE macro is allowed anywhere where code may be placed.

The <extent> on the DO parameter is the length of the branch to the corresponding ENDWHILE macro.

- **The ENDWHILE macro**

The ENDWHILE macro is only valid after a WHILE macro. It marks the end of the loop.

The REPEAT macro

The REPEAT macro is valid anywhere where code may be placed. The macro simply provides a label for the UNTIL test to branch back to.

- **The UNTIL macro**

The UNTIL macro is only valid after a corresponding REPEAT macro.

- **The FOREVER macro**

The FOREVER macro is only allowed after a corresponding REPEAT macro to provide an infinite loop (effectively an always FALSE version of the UNTIL macro).

B.8.6 CASE and associated macros

The macros SWITCH, CASE, ENDC, DEFAULT and ENDSWITCH allow a structure to be coded that selects one of a number of sequences of code depending on a control value. (The same effect could be achieved using IF, ELSEIF, ELSE, ENDIF but the CASE macros are often a more readable way to code a selection with many branches.)

The general structure is:

```
SWITCH<length> <ea>
{
        CASE[<extent>] <ea>
        .
        .
        [ ENDC[<extent>]   ]
}
[
        DEFAULT
        .
        .
        [ENDC[<extent>]   ]
]

ENDSWITCH
```

■ The SWITCH macro

The SWITCH macro introduces a set of CASE options. It is allowed anywhere where code is allowed. The <length> is the length of all the comparisons generated by all the relevant CASE macros.

The <ea> is the control variable which is used to select the CASE to be executed.

■ The CASE macro

The CASE macro introduces an option. The CASE macro is allowed only after a previous CASE, SWITCH or ENDC macro.

The <extent> on the CASE macro is the length of the branch to the next CASE, DEFAULT or ENDSWITCH macro.

The code following the CASE macro up to the next ENDC macro is executed if the control variable is equal to the <ea>, after which the program resumes at the ENDSWITCH macro.

■ The ENDC macro

The ENDC macro is allowed only after a previous CASE or DEFAULT macro. The <extent> is the length of the branch to the ENDSWITCH macro.

■ The DEFAULT macro

The DEFAULT macro is allowed only after a previous CASE or ENDC macro. Its purpose is to provide the default option if all previous options have not been satisfied.

The ENDSWITCH macro

The ENDSWITCH macro is only allowed after a previous ENDC, DEFAULT, CASE or SWITCH macro. Its purpose is to terminate the SWITCH structure.

B.8.7 Stack handling

The PUSH$ and POP$ macros save values (usually registers) on the A7 stack and restore them.

```
PUSH$[<length>]   <ea>
```

generates:

```
MOVE<length>   <ea>,-(A7)
```

and:

```
POP$[<length>]   <ea>
```

generates:

```
MOVE<length>   (A7)+,<ea>
```

B.8.8. STRING$ definition

```
<name>     STRING$     '<string>'
```

creates a string in standard format consisting of the two-byte length of the string followed by the characters of the string. The parameter should be enclosed in quotes, e.g.:

```
FILE          STRING$     'MDV1_MY_FILE'
```

and if it includes spaces, commas etc. the curly brackets must also be used, e.g.:

```
STRING1      STRING$     {'This is a string'}
STRING2      STRING$     {'A line followed by a line feed',$OA}
```

B.8.9 SUBROUTINE$ and associated macros

The macros SUBROUTINE$ and END$ are useful to mark the start and end of subroutines. If a subroutine needs to preserve a set of registers then these macros will generate the necessary MOVEM instructions; END$ also generates RTS.

In addition these macros will check that structure macros used inside the subroutine have been terminated properly, e.g. a missing ENDIF will cause a warning message when the END$ is processed.

```
SUBROUTINE$     <name>[,<multireg>]
         .
         .
         .
END$[<name>]
```

The <name> is generated as a label by SUBROUTINE$ so that it can be used as the destination of JSR and BSR instructions. If a <name> is coded on the END$ macro it is checked to see that it matches the previous SUBROUTINE$ and an error message is generated if not.

B.8.10 Macros for calling Qdos

The following macros are provided for making calls to Qdos:

```
QDOSMT$           <function name>
```

is used for manager traps and generates:

```
MOVEQ             #<function name>, D0
TRAP              #1
```

and:

```
QDOSOC$           <function name>
```

is used for open and close channel calls and generates:

```
MOVEQ             #<function name>, D0
TRAP              #2
```

and:

```
QDOSIO$             <function name>
```

is used for other input/output operations and generates:

```
MOVEQ               #<function name>,D0
TRAP                #3
```

In each case the <function name>s needed for the various Qdos traps are provided in the pararneter files supplied with the macro assembler, so if you INCLUDE the necessary parameter files and macro library you can write Qdos calls like this:

```
MOVEQ       #-1,D1              current job
CLR.L       D3                  old, exclusive
LEA         FILENAME,A0         name of
QDOSOC$     IO.OPEN             file to open
.
.
FILENAME       STRING$      'MDV1_INPUT_FILE'
```

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 a SECTION 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 bracket**
  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 – internal error – expression 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.

- **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 DS) 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, non-zero, value (eg the number of elements for DCB).

- **1A – 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 – read again the description of the instruction or directive involved.

- **1D – 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.

- **1E – invalid size**
  The size specified on the instruction or directive is not legal.

- **1F – size .B illegal for Areg**
  Byte operations on address registers are not allowed.

- **20 – label illegal on this directive**
  Many directives (eg INCLUDE, SECTION, LIST, PAGE) do not accept a label.

- **21 – too many errors**
  If a line has more than ten errors or warnings, only the first ten are printed, followed by this message.

- **22 – 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 Branch.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, REG, MACRO and MACLAB require a label

- **2A – 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.

- **2B – 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.

- **2E – not allowed with –NOLINK option**
  Many of the directives relating to relocation and linking may not be used if the –NOLINK option has been coded on the command line.

- **2F – not allowed in this context**
  This line is not allowed here because the context is wrong; this usually means that the wrong sort of location counter is in use, for example instructions are not allowed in a COMMON section. The most frequent cause of this error message is forgetting to code an appropriate SECTION directive.

- **30 – same used in SECTION and COMMON**
  You cannot have ordinary SECTIONs and COMMON sections with the same name.

- **31 – wrong relocation for PC-relative address**
  The assembler is trying to generate a PC-relative address but can't because the relocation factor of the instruction does not match that of the destination (e.g. a reference to a relocatable address from an instruction which follows an ORG).

- **32 – COMMON block name cannot be used here**
  The name of a COMMON block can only be used in a very restricted set of circumstances; this isn't one of them.

- **33 – same name used in SECTION and COMMON**
  You cannot have ordinary SECTIONs and COMMON sections with the same name.

- **34 – illegal expression for RORG**
  The <expr> in a RORG directive is not absolute and is not simple relocatable with respect to the current section (or has something else wrong with it). See the description of RORG for the full list of restrictions which apply to this <expr>.

- **35 – references to external symbols not allowed**
  Some directives need to know the actual values of their operands at assembly time and these do not therefore permit external symbols (those whose names appear in XREF directives) to be coded.

- **36 – expressions needing linking not allowed**
  In some circumstances expressions whose final value must be determined by the linker are not allowed; replace the expression with one whose value is known to the assembler.

- **37 – this symbol invalid in XDEF**
  There are various restrictions on the type of symbol that may be named in an XDEF directive. See the description of the XDEF directive for full details.

- **39 – label not found for GOTO**
  The label specified as the destination of the IFSTR, IFNUM or GOTO directive is not defined on a MACLAB directive in the current macro.

- **3A – not currently in a macro**
  This directive or function may only be used within a macro.

- **3B – user generated error**
  An ERROR directive was processed.

- **3C – expression does not result in a value**
  An expression used at this point must evaluate to an absolute value involving no forward references or relocation bases or external symbols.

- **3D – illegal parameter number for .PARM**
  The value of the expression lies outside the range 1 to .NPARMS.

- **3E – unexpected end of file after continuation line**
  The last line in a file ended in a backslash '\' and a continuation line was expected.

- **3F – MACRO name same as instruction or directive**
  You cannot define a macro with the same name as an instruction or an assembler directive.

- **40 – built-in function not allowed here**
  A function call is not allowed here (e.g. as the parameter to the .DEF function).

C.2 Warning messages

- **50 – size missing, W assumed**
  No size was specified on an index register.

- **51 – size missing, W assumed**
  The instruction or directive can have more than one size, but no size was specified.

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

- **53 – 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.

- **55 – 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.

- **56 – 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).

- **57 – 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.

- **58 – END directive missing**
  An END directive is expected at the end of the assembly, but end-of-file was found instead.

- **59 – XREF.S value will probably overflow when linked**
  An expression is of type XREF.S and is being placed in a 1-byte field. Depending on the actual value of the external symbols when the program is linked this may or may not cause an overflow.

- **5A – XREF.L value will probably overflow when linked**
  An expression is of type XREF.L and is being placed in a 1– or 2-byte field. Depending on the actual value of the external symbols when the program is linked this may or may not cause an overflow.

- **5B – ENDM directive missing after macro definition**
  End of file was found while processing a macro definition. This probably indicates that you omitted an ENDM directive or coded it in such a way that it could not be recognised (recall that ENDM must not be generated by variable substitution).

- **5C – ENDM directive missing while expanding a macro**
  The assembler ran off the end of the file while expanding a macro.

- **5D – user generated warning**
  A WARNING directive was processed.

- **5E – multiply defined symbol**
  Either the same name is used for two different macros or the same name is declared twice in LOCAL directives in the same macro. In both cases the first definition takes effect and subsequent definitions are ignored.

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. INCLUDEd) 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.

QL·Linker

Contents

1. Introduction

It is usually convenient, except in the case of very small programs, to write programs as several separate source files and compile or assemble them at different times. It is then necessary to combine these compiled parts of program to form a single program file before the program can be run and the program which does this combining is called a **linker**.

This manual tells you how to use QL Linker which has been produced for the Sinclair QL by GST Computer Systems Limited.

It tells you:

- how to load and run the linker
- what inputs the linker takes and what outputs it produces
- details of the Sinclair relocatable binary file format.

Note that for a compiler to be compatible with this linker it must generate its output in the official Sinclair relocatable binary format. You may write programs in several parts in different languages and link them together with QL Linker as long as all the compilers involved generate Sinclair relocatable binary format.

All Sinclair compilers for the QL will generate the correct output format and are compatible with this linker. It is possible however that other compilers not supported by Sinclair do not generate the appropriate format and are not compatible with QL Linker or other official Sinclair products.

1.1 Notation used in this manual

This section describes the notation used throughout the manual to describe syntax of all 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

<hexadecimal number> = $<hex digit>{<hex digit>}

<hex digit> = 0|1...|8|9|A|B|C|D|E|F

means that a hexadecimal number is a '$' sign followed by a hexadecimal digit, followed by any number of further hexadecimal digits, where a hexadecimal digit is any of the characters '0' to '9' or 'A' or 'B' or 'C' or 'D' or 'E' or 'F'. Some examples of hexadecimal numbers are $0, $4AFB, $000000.

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

<define command> = DEFINE <symbol> [=] <expression>

2. How to run the linker

The linker may be loaded and run in one of two ways:

- **Interactive mode**
  In this mode the linker will identify itself and prompt you for a command line. Upon completion of a link the linker will prompt you for another command line (unless a fatal error has occurred).

  You may run the linker in interactive mode by any of the following commands where `dev_` is the device from which the linker is to be loaded (which may be any storage medium).

  – To run in parallel with the SuperBASIC interpreter:

  ```
      EXEC dev_link (see Notes)
  or: EX dev_link (see Notes)
  ```

  – To wait for completion of the linker:

  ```
      EXEC_W dev_link
  or: EW   dev_link
  ```

- **Non-interactive mode**
  In this mode the linker receives its command directly from the SuperBASIC interpreter and does not interact with you. On completion of the link the linker will exit to allow the SuperBASIC interpreter to continue.

  You may run the linker in non-interactive mode by one of two commands (see Notes):

  – To run in parallel with the SuperBASIC interpreter:

  ```
  EX dev_link; "<command line>"
  ```

  – To wait for the link to complete:

  ```
  EW dev_link;"<command line>"
  ```

where <command line> is described in 2.2. The quotes around the command line are required for the SuperBASIC interpreter to accept the line.

- **Notes**

  The EX and EW commands are only available if you have a copy of QL Toolkit, and are not part of the standard SuperBASIC.

  The EX and EW commands allow you to pass data files to the program by specifying them after the program name. If any files are specified in this way they will be ignored by the linker. See the QL Toolkit documentation for information on the full use of the EX command.

2.1 Altering the window

If you wish to alter the screen window used by the linker you may do so by running the program WINDOW_MGR and answering the questions it asks.

2.2 Command line format

The format of the command line is:

```
[<module> [<control> [<listing> [<program>]]]] {<option>}
```

where:

```
<option> =
            –WITH <control> |
            –NOPROG | –PROG[<program>]|
            –NOLIST | –LIST[<listing>]|
            –NODEBUG |–DEBUG [<debug>]|
            –NOSYM | –SYM | –CRF |
            –PAGELEN <length> |
```

(the options may, of course, be in either upper or lower case and the case is not significant)

<module>	= <file name>	file name of an object file
<control>	= <file name>	file name of control file
<program>	= < file name>	file name for program output
<listing>	= <file name>	file name for listing output
<debug>	= <file name>	file name for debug output
<length>	= <digit> {<digit>}	number of lines per page.

2.3 Options

The options have the following meanings:

–WITH take the following name as the control file name. If this option is specified, the positional control file name is ignored.

–NOPROG do not generate a program file. If this option is in effect then the positional program file name is ignored.

–PROG generate a program file (default). If the optional <file name> is coded then the positional file name is ignored.

–NOLIST do not generate any listing output. If –NOLIST is coded then the positional listing file name is ignored.

–LIST generate a listing (default); if the option is followed by a <file name> then this is the name of the <listing> output and the positional <listing> parameter is ignored.

–NODEBUG do not generate a debug file (default).

–DEBUG generate a debug file. If the <file name> is given then it will become the debug file otherwise the file name will default.

The following options apply to the listing file only. They will be ignored if –NOLIST is in effect.

–NOSYM do not generate a symbol table listing in the listing file.

–SYM generate a symbol table listing. The listing will be alphabetically sorted with the value of the symbol with the section and module name in which it was defined (default).

–CRF generate a cross reference form of symbol table listing. If this option is requested a cross reference form of the symbol table is generated instead of the symbol table list.

–PAGELEN specify the number of lines per page for paginated output. If this option is not supplied the value will default.

If an option is followed by a file name (where applicable) the file name will override the corresponding positional file name (if given) on the command line. If an option specifies that a file will not be generated (–NOPROG, –NOLIST, –NODEBUG) then the file will not be generated even if a positional file name has been given.

Where conflicting options are given on the command line then the last option coded will take effect; for example:

```
–NOPROG –PROG MDV1_PROG
```

will produce a program file, whereas

```
–PROG MDV1_PROG –NOPROG
```

will not.

2.4 Command line processing

The minimum command line then just consists of the name of one module file. In this case the linker will generate a program file (whose name is constructed as below from the module name) and a full listing file (whose name is also constructed as below).

If no module file name is specified, but a control file name is given (after a –WITH option) then the program, listing and debug file names will be constructed as below based on the control file name.

2.4.1 Construction of output file names

If a module file name is given then the file name is examined. If the file name does not end in '_REL' then the full file name becomes the base file name, otherwise the file name with the '_REL' stripped off becomes the base file name.

If no module file name is given then the control file name is examined. If the file name does not end in '_LINK' then the full file name becomes the base file name, otherwise the file name with '_LINK' stripped off becomes the base file name.

The default names are then constructed from the base file name as follows:

1) The listing file name is the base file name with '_MAP' appended.

2) The program file name is the base file name with '_BIN' appended.

3) The debug file name is the base file name with '_DEBUG' appended.

If an output file name is given explicitly either as a positional parameter or in an <option> then the file name will override the corresponding default name. Any file name given explicitly must be given in full as the file name will be used exactly as entered.

2.4.2 Input file name defaults

The linker has two types of input file: the control file, which tells the linker what to do (if more information is needed than can be coded in the command line) and relocatable binary files, which are the output files from compilers and assemblers that contain the parts of the program to be linked.

The linker expects that control file names will usually end in '_LINK' and that relocatable binary file names will usually end in '_REL' and will find these files even if the final component is missing from the file name given on the command line or in the control file.

For a module file name (or library file name), if the module file name ends in '_REL' the linker will use the file name exactly as given. If the file name does not end in '_REL' then '_REL' will be appended to the file name; if an open error occurs on this file then the original file name is used instead (by stripping off the '_REL' again).

This defaulting will apply to all module input commands in the control file as well as to any relocatable binary file name given on the command line.

If the control file name ends in '_LINK' then the linker will use the control file name exactly as given. If the file name does not end in '_LINK' then '_LINK' is appended to the file name; if an open error occurs on this file then the original file name is used as the control file name.

2.5 Command line examples

```
MDV1_FRED
```

Take the file MDV1_FRED_REL as an object module and turn it into a program file called MDV1_FRED_BIN. The listing is called MDV1_FRED_MAP.

```
MDV1_MYPROG MDV1_PASCAL-LINK -NOLIST
```

Link the file MDV1_MYPROG_REL according to the instructions in MDV1_PASCAL_LINK. The program is called MDV1_MYPROG_BIN.

```
-WITH MDV1_FRED
```

Take MDV1_FRED_LINK as the control file, place the program in MDV1_FRED_BIN and place the full listing output in MDV1_FRED_MAP.

```
-WITH MDV1_FRED -LIST SER1 -NOPROG
```

Take MDV1_FRED_LINK as the control file, do not generate a program file but print the listing as it is produced.

```
-WITH MDV1_FRED -PROG MDV2_FRED_BIN
```

Take MDV1_FRED_LINK as the control file, place the program in MDV2_FRED_BIN and place the listing output in MDV1_FRED_MAP

2.6 Termination

When the link has finished, and if there have been no operating system errors, the linker will issue a message giving the status of the link. If the linker has been run interactively then the linker will repeat the prompt asking for a command line. You can now do another link without having to reload the linker. When you have done all the links that you want you may reply to this prompt with an empty command line and the linker will terminate.

3 Linker inputs and outputs

The linker uses the following inputs and outputs. The formats of these files produced are described in Appendix B and the format of the control file is specified in detail in section 4 below.

3.1 Command line input

When run interactively the linker will read a command line from the keyboard to tell it what to perform. Any errors in the command line will result in an error message followed by a reprompt of the command line. See section 2 above for full details of the command line.

3.2 Control file

If the command line includes a control file name the linker will expect as input a single text file containing a list of instructions to perform.

The text file may be on any serial device that can detect end of file (which terminates the input). Suitable devices are Microdrive or disk.

The control file is described in detail in section 4 below and a summary of it is included in Appendix B.

3.3 Relocatable object file inputs

The linker, on instruction from the command line or control file will read one or more relocatable object files (which may contain one or more object modules).

The files are opened for random access to allow modules to be extracted independently (for EXTRACT and LIBRARY commands) so that suitable devices for the files are Microdrive or disk.

The normal user of the linker will not need to know the details of the format of relocatable binary files. The specification of this format is however included in Appendix B for the benefit of advanced assembler programmers and compiler writers.

3.4 Screen output

The linker writes information to the screen to inform the user what is happening. This includes a start up message identifying the program, and a prompt for a command line.

The linker writes all error and warning messages to the screen and on completion of the link will print a summary of the number of errors and warnings and the number of undefined symbols (if any).

The linker tells you when it is starting to read the relocatable object modules. It does this twice. The second pass can be expected to take a lot longer than the first pass if listings and/or program output are wanted.

The linker finally gives a message indicating the completion status of the link and if run interactively reprompts for another command line.

3.5 Linker listing output

An optional linker listing will be generated, showing the commands used in the production of the link and a map of the layout of the executable file. The map will also show a list of all global symbols and their values and an optional cross reference giving the modules which reference them.

3.6 Program file output

The linker will optionally generate a program file which will be the result of combining the relocatable binary files. This is normally a file which can be run by the operating system as a program but it is possible to use the linker to produce files which cannot be run directly (e.g. files that are to be used for programming PROMs).

A relocation table, if produced by the linker, will be included within the program file. This is only necessary when using a compiler which does not generate position-independent code and full instructions for using this facility should be included in the documentation of such compilers.

3.7 Debug file output

The linker will optionally produce a symbol table file for use by a symbolic debugger program.

4 The control file

In the simplest mode of operation you may link a single input file by just giving the name of the input file on the command line. However the linker may accept more than one input file and may also accept more complex instructions for the generation of the output file. These instructions can be provided to the linker by a control file.

The control file is a text file which gives a series of instructions to the linker. The complete set of instructions to the linker will be given here for completeness; however you may not need to use all the instructions if you are just beginning in programming.

If you are programming in a high level language (Fortran, Pascal or C) there may be either a standard control file for linking your module with the library for the language or a template file to give you instructions on how to link your file to make a program. Please consult the documentation of the language concerned for more information on this topic.

Unlike the command line input, the control file input is not interactive and any errors in the control file will cause the link to be abandoned.

All letters in control file commands and command parameters may be in either case and case is not significant.

4.1 Comments in the linker control file

The linker accepts comments in the linker control file to explain to the user what a particular control file does. A line will be considered a comment if the **first** character in the line is an asterisk (*), semicolon (;) or an exclamation mark (!). A blank line is also considered to be a comment.

The use of comments in a control file may assist you in editing the control file to suit your particular program.

e.g.

```
*  Example template file for linking
*  together modules under language L. It is
*  NOT a template file for any particular
*  language and should not be taken as such


*  Step 1 - initialisation.
*  ========================
*
*  Language initialisation must be included
*  first.
   INPUT  MDV1_LINIT_REL

*
*  Step 2 - system interface library
*  ===================================
*
*  system interface library - only
*  include if your program is trying to
*  access system routines directly. (by
*  uncommenting the line).
*
*  INPUT  MDV1_LSYSLIB_REL

*
*  Step 3 - user modules
*  ======================
*
*  For each module that you wish to include
*  in the link include a line here of the form
*
*  INPUT <filename>

*
*  Step 4 - language library
*  ===========================
*
*  Language library - must be included at all
*  times.
*
   LIBRARY  MDV1_LLIB_REL
```

4.2 Module input commands

There are three commands to instruct the linker to input modules from relocatable binary files. These are INPUT, EXTRACT and LIBRAY.

4.2.1 INPUT <file name>

This command instructs the linker to read the file named and place **all** modules encountered in the file into the link. Include one command for each file that you wish to include in the link.

e.g.

```
INPUT MDV1_FILE1_REL
INPUT MDV1_FILE2_REL
```

will include all modules in MDV1_FILE1_REL and MDV1_FILE2_REL which may be separate routines created by a compiler.

A special case of the input command is the command

```
INPUT *
```

which instructs the linker to use the input module name given on the command line as the file name to input. This feature allows the generation of a template file which can be used to link a single module output from a compiler with all the required libraries for the high level language. The template file is then used by a command line of the following form (the –WITH is optional):

<module file name> [–WITH] <template file name>

e.g.

```
*
*   example template file for the language
*   L with an initialisation module called
*   MDV1_LINIT_REL and a language library
*   called MDV1_LLIB_REL
*
*   start with the initialisation routines
*
```

```
INPUT MDV1_LINIT_REL
*
*  now include the user module (from the
*  command line)
*
INPUT *
*
*  now include all modules from the language
*  library
*
INPUT MDV1_LLIB_REL
```

4.2.2 EXTRACT <module name> from <library file name>

This command instructs the linker to search the library file name given for the module requested. If the module is found it is included in the link. If not an error message is generated and the link is aborted.

Include one extract command for each module that you wish to explicitly include from the library file.

e.g.

```
*
*  example control file for the language L
*  with an initialisation module called
*  MDV1_LINIT_REL
*  and a language library called
*  MDV1_LLIB_REL
*
*  the example has now been modified to
*  extract the required initialisation
*  module from the library (which may
*  contain many initialisations for
*  different purposes).
*
```

```
*  start with the initialisation routines
*  (only need the first routine)
*
EXTRACT LINIT FROM MDV1_LINIT_REL
*
*  now include the user module
*
INPUT *
*
*  now include all modules from the language
*  library
*
INPUT MDV1_LLIB_REL
```

4.2.3. LIBRARY <library file name>

This command instructs the linker to search the library file named from start to finish for modules which satisfy any currently unresolved references in the link. When a module is found which satisfies an unresolved reference it is included in the link and the library search continues from the current position.

The use of this form of a library search means that the ordering of modules within a library may be important, as a module read in to resolve a reference may itself generate another unresolved reference, which then may cause a module following to be read in. Note that the library is searched only once for a library command. If the library is to be rescanned then this is achieved by including another library command specifying the same library file name.

You should include one library command for each library that you wish to search.

e.g.

```
*
*  example control file for the language L
*  with an initialisation module called
*  LINIT
*  and a language library called
*  MDV1_LLIB_REL
*
*  The example has now been modified to
*  extract the required
*  initialisation module from the library
*  (which may contain many initialisations
*  for different purposes).
*  As the library command will only extract
*  modules which satisfy currently
*  unresolved references, the initialisation
*  routine may now be included in the same
*  language library, as the module will not be
*  read again since it already satisfies
*  all references that it can possibly
*  resolve.
*
*  start with the initialisation routines
*  (can include init in same library)
*
EXTRACT  LINIT  FROM  MDV1_LLIB_REL
*
*  now include the user module
*
INPUT  *
*
*  now include all modules from the language
*  library which are required to satisfy any
*  unresolved references
*
LIBRARY  MDV1_LLIB_REL
```

4.3 Space allocation commands

The previous section described the commands for determining which input modules are to be included in the link. This section describes briefly how the linker allocates space for the modules in the output file and the linker commands which may affect this allocation.

Normally the default space allocation methods are adequate and the user writing normal applications programs will not need to use any of the commands described in this section (except that some may be necessary in template files supplied with particular compilers).

Initially the default allocation mechanism will be described and later the effects of each command on this allocation mechanism will be considered.

As programs may be loaded and executed anywhere in memory they must be written in position independent code. Therefore references in the following sections to low addresses and start addresses are referring to their positioning in the program file and not to their position in memory when run.

Generally an object module consists of either Absolute sections and/or at least one Relocatable (or Common section). The allocation of each section type is as follows:

- **Absolute sections**

  Absolute sections are allocated space first in the output file from their start address (relative to the start of the file). The linker will issue a warning if any absolute sections overlap in the link.

- **Relocatable sections**

  As each input module is read in turn (as ordered by the INPUT, EXTRACT and LIBRARY commands) the linker builds up a list of relocatable sections in the order in which they are first encountered.

Once the sizes of each relocatable section is known then the allocation of space is made such that each relocatable section starts at the lowest possible address following the previous relocatable section while avoiding any absolute sections already allocated. The start of each relocatable section is word aligned.

- **Common sections**

  By default common sections are treated as relocatable sections except for the following differences.

  - Each common section is placed in the list of relocatable sections when a COMMON directive is encountered (instead of SECTION directive)

  - If a COMMON directive references a section already used in a previous module then a subsection is created which starts at the same address as the section start (i.e. overlayed). The size of the common section is then the maximum of the subsection sizes.

4.3.1 Effect of commands on space allocation

The following commands alter the mechanism by which the linker allocates space for each section.

- **SECTION <section name>**

  This command names a section to be included in the link. The effect on the storage allocation is that named sections are allocated space first in the order declared with any unnamed sections allocated space following (as with the default case).

- **COMMON <common option>**

  The COMMON command instructs the linker how to allocate space for common sections. In the default case common sections are treated as if they were relocatable sections for the purposes of address allocation. However the following options are available:

- **END**

  This option instructs the linker to allocate space for common sections after all relocatable sections have been placed. This means that the common sections appear at the high end of the memory allocation.

  If any common sections have been named by a SECTION command then they are allocated space first followed by the common sections as encountered in the input files. The allocation of common sections is such that they avoid any absolute sections as with the normal relocatable sections.

- **DUMMY**

  This option instructs the linker to build a separate allocation for common sections. The allocation starts from address zero and ignores any allocation taken by relocatable or absolute sections.

  The linker will use the dummy allocation to resolve global symbols in common sections relative to the start of the common area, so that a run time system can allocate memory separate from the program for the purposes of storing common. The global variables are then used as offsets from the start of the common region.

  Note that with this option no space is made in the program file for the common sections, so they may not be initialised. Any attempt to place data bytes in the common regions with this option in effect will cause an error.

■ **RELOC <section name>**

This command is only necessary when the linker is used to link output from compilers which do not generate position-independent code. The instructions supplied with the compiler for the use of this command should be followed.

This command if present instructs the linker to generate run time relocation information and to place the information in the SECTION named. The command declares the section (as with a section command) so that any sections which must come before this section must be named in SECTION commands before the RELOC command is given.

The run time relocation table is placed at the end of the section named, so that any data from the relocatable binary files for the same section will be placed in front of the run time relocation table in the order encountered.

The command also declares the section to be normal relocatable so that any attempt to declare the section as a common section will result in an error.

- **OFFSET <value>**

This command instructs the linker to start the allocation of section starting at the value given instead of at address 0. The value may be decimal or hexadecimal (starting with a '$' character) and is unsigned. The value is written into the spare four bytes of the information section of the file header of the program file (see Appendix B).

The effect on the allocation of space is as follows:

– **absolute sections**

The allocation of absolute sections is not affected. However any absolute sections which start below this address are not written to the output file and a warning message is output.

– **relocatable sections**

Relocatable section allocation begins from the address given in the OFFSET command instead of at zero.

– **common sections**

If COMMON DUMMY is in effect then the allocation of common sections starts from address 0 regardless of the value given in the OFFSET command. For all other COMMON options the allocation is as described under the COMMON command.

4.4 Defining symbols at link time

Normally symbols that the linker knows about are declared and given values from within relocatable binary files. Sometimes, however, it is useful to be able to define symbols from the linker control file; examples of why this might be useful are:

– a subroutine name has accidentally been spelt differently in two different modules; as a temporary fix (until one of them is recompiled) the two symbols can be made equivalent using the DEFINE command

– a set of subroutines have not been written yet but it is desired to test the part of the program that has been written; the missing symbols can be made equivalent to an error routine with the DEFINE command

– a number contained in a library module, such as a memory requirement figure, may need to take different values in different links; these values may be assigned with the DEFINE command.

■ **DEFINE <symbol>[=]<expression>**
The linker allows you to define symbols at link time rather than needing to declare all symbols in relocatable object modules. The define command also allows expressions with the following syntax:

```
<expression> = [ – ] <term> { <op> <term> }
<op> = – | +
<term> = <symbol> | <value>
<value> = <digit> { <digit> } | $<hexdigit> { <hexdigit> }
<hexdigit> = <digit> | A | B | C | D | E | F
```

A symbol used in the expression side of the DEFINE command may be a reference to a symbol in a relocatable binary file or a reference to a previous symbol defined by a define command. A forward reference to a symbol to be defined by a future define command is illegal and will produce an error message. The symbol named in the DEFINE command may not also be used in the expression.

If a symbol used in an expression remains undefined after all modules have been read in a warning is issued by the linker. The value of the DEFINEd symbol is then undefined.

e.g.

```
DEFINE SCREEN = $28000
DEFINE MAXPAR = 10
DEFINE USERSPACE = 1000
DEFINE TOTALSPACE = LOCAL+GLOBAL+USERSPACE
```

(where LOCAL and GLOBAL are declared in relocatable object modules)

4.5 The DATA command

- **DATA<value>[K]**

The DATA command specifies the amount of data space to reserve for a program for the stack and heap. The value may be decimal or hexadecimal. This value is written to the header of the program file and is used by the operating system to allocate room for the stack and heap. The value may be specified in bytes or Kbytes (1024 bytes)

The data space requirement is also read from the header of the input files for each module to be included in the link (see below). In this case the **maximum** of the data requirements of the input modules is taken as the data requirement unless the value specified in the DATA command is larger.

Note that the linker checks the type of file for each input file. The file type is contained in the file header and can currently take the following values (other types may be added later).

0	text file.
1	executable program file.
2	relocatable binary file.

The data space requirement is used only if the input file is of type 2 (relocatable binary).

While the linker will accept any file type as a relocatable binary file it will ignore the data space requirement of any file which is not type 2. All official Sinclair assemblers and compilers will generate relocatable binary output files with the file type correctly set to 2.

5 Actions of the linker

This section gives a brief description of how the linker functions and the expected actions when errors are encountered. The linker functions are split into many phases which are logically separate although each phase may use information extracted from previous phases.

5.1 Command line validation

In this phase the linker reads the command line and decides which input and output files to use. If the command line contains any errors the linker will display an error message stating the problem and will reprompt for another command line.

If the command line is valid the linker will attempt to open all output files requested and the linker control file (if a name is supplied). If the opening of any files fail the linker will give a message indicating the problem and will reprompt for another command line.

If the linker is run interactively it will reprompt for another command line. If not then the linker will display a message indicating an invalid command line supplied and exit.

5.2 Control file validation

If a control file name is given the linker will read the control file line by line validating each command in turn. If any errors are reported at this stage the linker will report the error but continue reading the control file.

If any errors occur in the control file the linker will not perform the link but will reprompt for another command line.

5.3 Pass 1 of relocatable object modules

If the command line and control file (if given) contain valid commands the linker will issue a message saying `starting pass 1` and will read all the relocatable object files requested and determine the sizes of each section to be placed in the output file. During this pass the linker will issue error and warning messages as appropriate to indicate any problems encountered.

If it fails to open any requested input files or encounters any errors during this pass the linker will issue an error message stating the problem and will continue processing the rest of the input files.

At the end of pass one if any errors have been encountered the linker will prompt for another command line. If only warnings have been detected the linker will continue with the link.

5.4 Between pass processing

After pass 1 the linker determines where to place everything in the program file and resolves all global symbols. The load map is generated at this time along with a list of all absolute, user defined and undefined symbols.

5.5 Pass 2 processing

During this pass all the relocatable objects modules are reread and the program file created. If any errors are encountered at this stage the link is aborted.

5.6 Post processing

After pass 2 the symbol table is written out and if required a debug file is created. Upon completion of the symbol table the linker issues a summary message stating the numbers of errors and warnings and the number of undefined symbols. The linker then reprompts for another command line. Entering a blank line at this stage terminates the linker.

Appendix A – Error and warning messages

This appendix lists the error and warning messages which can be produced by the linker in the phases in which they should be encountered.

A.1 Command line errors

The linker on encountering an error in the command line will display a message indicating the problem and reprompt for another command line. It will not attempt to parse the line following the error.

- **ERROR – 01 File name too long – <file name>**
  Either a file name entered on the command line or a default file name generated from the primary file is too long. The full Qdos file name can only be 44 characters long.

- **ERROR – 02 No link file given with the –WITH option**
  A –WITH option has been entered without a link file name. The –WITH option must be followed by a file name.

- **ERROR – 03 Page length missing following –PAGELEN option**
  The –PAGELEN expects a value to set the page length for formatting on a printer.

- **ERROR – 04 Page length is not a number**
  The item following the –PAGELEN option is not a number.

- **ERROR – 05 Page length too small. Minimum is 20 lines**
  As the listing output is formatted with headers, titles and subtitles the minimum realistic page length is 20 lines.

- **ERROR – 06 No input module or control file given**
  The linker requires as input either a module file name or a control file name. If neither is given then the linker does not have any input files to act upon.

- **ERROR – 07 Illegal option given on command line <option>**
  An unrecognised option has been entered. The option parameter indicates which option the linker was unable to recognise.

A.2 Control file errors

The linker will on encountering an error in the control file list the line for which the error has occurred and print a message indicating the cause of the error. The linker will process the rest of the control file but will not proceed with the link.

- **ERROR – 09 Illegal or unrecognised command <command>**
  An illegal or unrecognised command has been encountered in the control file. The command parameter is the command that the linker failed to recognise.

- **ERROR – 0A Too many parameters <parameter>**
  The linker has encountered too many parameters on the line. The command has been processed but the link will not be performed.

- **ERROR – 0B Not enough parameters, expecting <item>**
  The linker did not find enough parameters on the line. The item parameter indicates which item was expected which will be one of the following:

Item	**Command**
file name	INPUT, EXTRACT or LIBRARY
module name	EXTRACT
FROM keyword	EXTRACT
section name	SECTION
END or DUMMY	COMMON
value	OFFSET or DATA
symbol name	DEFINE
expression	DEFINE

- **ERROR – 0C No module name given in command line for INPUT ***
  The linker has encountered an INPUT * in the control file but no module name was given on the command line.

- **ERROR – 0D FROM keyword missed out or incorrectly spelt**
  In an extract command the FROM keyword was not found. This keyword must be present.

- **ERROR – 0E Section already exists <section>**
  The section named in the section command has already been named in a previous SECTION or RELOC command and so cannot be placed in the order requested.

- **ERROR – 0F Illegal option, DUMMY or END only allowed**
  An illegal common option has been given. The linker only recognises the keywords DUMMY and END.

- **ERROR – 10 Only one COMMON command allowed**
  Only one common command is allowed in any one link.

- **ERROR – 11 Symbol was used in DEFINE command: <symbol>**
  A symbol being defined in a DEFINE command has already been used in a previous DEFINE expression. Forward referencing of defined symbols is not allowed.

- **ERROR – 12 Symbol is being redefined <symbol>**
  The symbol being defined has already appeared in a previous DEFINE command and cannot be redefined.

- **ERROR – 13 Syntax error in DEFINE command <expression>**
  The linker has detected an error in the syntax of the DEFINE command. The expression following the error message starts from the character position which caused the syntax error.

- **ERROR – 14 Only one RELOC command allowed**
  Only one RELOC command is allowed in a link. It is meaningless to try to place the run time relocation information in more than one section.

- **ERROR – 15 OFFSET value is not a number**
  The value following the OFFSET command is not a number.

- **ERROR – 16 Only one offset value is allowed**
  As the OFFSET value is the start point for allocation of memory for the program only one value is allowed.

- **ERROR – 17 DATA value is not a number**
  The value entered following the DATA command is not a number. The DATA value can only be a number, an expression is not allowed.

- **ERROR – 18 Only one DATA value allowed**
  The DATA value specifies the amount of memory to be reserved for data space by Qdos when the program is initially run. Only one DATA command is allowed in any one link.

A.3 Low level errors

These errors are detected when parsing the line at a low level. The error messages are followed by a message indicating which command was being processed at the time the error was encountered.

- **ERROR – 19 Numeric overflow**
  The numeric value following an OFFSET or DATA command is too large to fit in a 32 bit word.

- **ERROR – 1A Syntax error in number**
  The linker has detected an illegal character while processing a number. This is normally caused by a $ which is not followed by a hexadecimal digit.

- **ERROR – 1B Invalid character**
  The linker has detected an illegal character while processing a line.

- **ERROR – 1C Decimal number overflow**
  The linker has detected that a decimal number has overflowed to negative.

A.4 Processing errors and warnings

These errors are detected while processing the link after validation of all command inputs to the linker. The description of the error messages are followed by a description of the actions performed following the error. Warning messages always result in the linker continuing from the current position in the link.

- **ERROR – 1D EXTRACT – module not found**
  The linker could not find the module requested in an extract command in the file specified. The linker will continue to process all remaining inputs in pass 1 and then prompt for another command line. The program file will not be produced.

- **ERROR – xx Error in relocatable binary file <file name>**
  This error message indicates a problem with the relocatable binary file provided to the linker. The linker will continue to process all remaining input files in pass 1 and then prompt for another command line. The program file will not be produced.

- **ERROR – 2D Attempt to initialise dummy COMMON in <file>**
  The linker has detected an attempt to place data into a COMMON section with the COMMON DUMMY option in effect. As no space is saved for the COMMON blocks they may not be initialised in this way. The linker will continue to process all remaining input files in pass 1 and then prompt for another command line. The program file will not however be produced.

- **ERROR – 2E Absolute section below OFFSET address in <file name>**
  This error indicates that an OFFSET command has been given in the linker control file but an absolute section resides below the OFFSET address. The linker will continue but the part of the section below the OFFSET value will not be contained in the file.

- **ERROR – 31 Phasing error occurred in <file>**
  The linker has encountered a phasing error either in processing of the relocatable binary files in pass 2 or when evaluating a DEFINE expression. This error should not occur.

- **ERROR – 32 Out of memory**
  The linker has run out of memory while trying to allocate more memory for internal tables. The linker will exit after printing this message.

- **ERROR – 33 Attempt to allocate large record**
  The linker has attempted to allocate a record which is larger than the current memory allocation. The linker will exit after printing this message. This should never occur.

- **ERROR – 34 Incompatible section type for section <section>**
  This error indicates that a section has been used both as a normal relocatable section and as a COMMON section. The linker will process all remaining input files in pass 1 however no program file will be produced.

- **WARNING – 35 Insufficient memory for cross reference**
  This message indicates that the linker cannot allocate sufficient memory for the cross reference listing. The linker will continue but a normal symbol table listing will be given instead of a cross reference.

- **WARNING – 36 Truncation error at offset <offset>**
  This warning indicates that a value has had to be truncated to fit into a byte or word expression. The offset value gives the location at which the truncation has occurred. The linker will continue, however the program may encounter problems if run.

- **WARNING – 37 Undefined symbol was used in DEFINE expression:**
  This warning indicates that a symbol which was used in the expression part of a DEFINE command is still undefined. This means that the result of the DEFINE command is also undefined.

- **ERROR – 3A Internal error**
  The linker has detected an internal error (consistency check). This error should never occur.

- **WARNING – 3B Multiply defined symbol <symbol>**
  This warning indicates that a symbol has been defined more than once in the link. The first value encountered will be the value used by the link.

- **WARNING – 3E Abs section overlaps next one in <file>**
  This warning indicates that two absolute sections overlap each other in the program file. This means that the second absolute section will overwrite the first.

A.5 Operating system errors

When the linker gets an error code from Qdos the action taken is dependent on what the linker is trying to do when the error is encountered. The linker will take the following action on encountering errors:

- **Open errors on files**
  These errors are reported by the linker. If the error occurs on opening the program, listing, debug or control file the linker will reprompt for a command line. If an error occurs on opening a relocatable object file the linker will continue until the end of pass 1 to validate that all other files may be opened.

- **Read and write errors on files**
  If the linker encounters a read or write error on a file (other than end of file on read) the linker will report the error and exit.

- **Close errors on files**
  If the linker encounters an error on closing files the linker will report the error and continue.

In most cases the linker will display the file name which caused the error except in the case of a read error on a module file where the linker does not display the name of the file which caused the error.

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

Appendix B – File formats

This appendix describes the format of output files produced by the linker.

B.1 Summary of control file commands

This section is a quick summary of the commands possible in the linker control file.

Lines beginning with *; or ! are comments and are ignored by the Linker. All letters in the control file input can be in either case and case is not significant.

- **INPUT <file name>**
  Instructs the linker to include all modules from the named file in the link.

- **EXTRACT <module name> FROM <file name>**
  Instructs the linker to find the module named in the file. If the module is found it is included in the link.

- **LIBRARY <file name>**
  Instructs the linker to search the library from start to finish. Any modules in the library which satisfy any currently unresolved references are included in the link.

- **SECTION <section name>**
  Declares a section to the linker. All declared sections are allocated space before any undeclared sections.

- **COMMON <common option>**
  Instructs the linker how to handle COMMON sections (if any are encountered).

- **RELOC <section name>**
  Instructs the linker to collect run time relocation information and place it in the section named.

- **OFFSET** <**value**>
  Instructs the linker to start address allocation and to write the output file from the address given in the value parameter.

- **DEFINE** <**symbol**> **[=]** <**expression**>
  Defines a symbol at link time. If the expression includes a symbol which has not already been defined then the linker expects to find it in a relocatable object module.

- **DATA** <**value**> **[K]**
  Defines the amount of data space required by the program when it is run.

B.2 Relocatable binary format

This section defines the official Sinclair relocatable binary format. It is self-contained and uses some terms with different meanings from those used in the rest of the linker manual.

A relocatable object file consists of a sequence of modules, each of which is a sequence of bytes terminated by an END directive (see below). It should have a Qdos file type of 2 though this will not be enforced by the linker. Interspersed with the sequence of bytes can be directives from the list below; a directive is a sequence of bytes beginning with the hex value FB.

When otherwise unmodified by a directive, a byte indicates that it should be inserted at the current address and the address should be stepped by 1. The special directive FB FB inserts the value FB in this way.

Note that bytes are **overwritten** on (not added into) the byte stream, so that if several sections are located at the same address, it is possible to overlap (or even interleave) their contents. This is useful for Fortran block data.

In the following syntax definition, <words>s and <longword>s need not be word aligned: they just follow on from the preceding data with no padding bytes.

A <string> consists of a length byte (value range 0–255), followed by the bytes in the string. A <symbol> is a <string> of up to 32 chars. A symbol should start with a letter (A–Z) or a dot and the other characters may be letters, digits, dollar, underline or dot.

B.2.1 Definition of a SECTION

A SECTION is a contiguous block of code output by the linker. Each section has a name, and any source file can add code to one or more of the sections. A module's contribution to a section is called a subsection.

The linker will arrange that each section or subsection will start on an even address, by inserting one padding byte if necessary. The value of this byte will be undefined.

Note that if a module returns to a section, this is part of the same subsection and the linker will **not** re-align on a word address.

When a section name is used in an XREF command the address of the start of the subsection is used.

Note that section names are maintained separately from symbol names (and module names), so there can be a section, a symbol and a module all with the same name without any danger of confusion.

B.2.2 Directives

B.2.2.1 SOURCE Syntax: FB 01 <string>

The <string> in this directive indicates information about the source code file from which the following bytes were generated. This directive should only appear at the start of a module (ie at the start of the file or immediately after an end directive: see section B.2.3).

The string will start with the **module name** which may be followed by a space followed by a field of further information about such things as the version number or the date of creation or compilation. The string should contain only printable characters and be no longer than 80 characters.

This **module name** should conform to the syntax of a <symbol> defined above, and may be used by the linker to identify individual modules within a library (see section B.2.4). The module name can be generated from a Qdos filename, but if so it is recommended that the Qdos device name is first stripped off.

B.2.2.2 COMMENT Syntax: FB 02 <string>
The <string> in this directive is a line of comment. It will have no effect on the binary file, but should be included at some suitable point in a link map. The string should contain only printable characters and be no longer than 80 characters.

B.2.2.3 ORG Syntax: FB 03 <longword>
This indicates that the bytes following the directive are to start at the absolute address given in the parameter. This applies until the next ORG, SECTION or COMMON directive.

B.2.2.4 SECTION Syntax: FB 04 <id>
This indicates that the bytes following the directive are to be placed in the relocatable section whose name was defined in a DEFINE command with the id value specified. See B.2.2.8.

This applies until the next ORG, SECTION or COMMON directive.

B.2.2.5 OFFSET Syntax: FB 05 <longword>
This directive updates the output address: the longword specifies the address relative to the start of the current subsection or the latest ORG directive.

The parameter is unsigned, so the offset may not be negative.

B.2.2.6 XDEF Syntax: FB 06 <symbol> <longword> <id>
This indicates that the symbol whose name is the <symbol> is defined to be the value given in <longword>, relative to the start of the subsection referred to by the <id>. Note that an <id> of zero defines the symbol to be absolute.

See section B.2.2.8 for definition of <id>

B.2.2.7 XREF
Syntax: FB 07 <longword> <truncation-rule> { <op> <id> } FB

This indicates that the result of an expression involving user symbols or other relocatable elements is to be written into the byte stream. Note that this command does not overwrite some existing bytes, but appends new bytes to the output.

The <longword> parameter defines an absolute term for inclusion in the expression to be evaluated by the linker.

The <truncation-rule> parameter is a byte which defines the size of the final result and the circumstances in which the linker might give a truncation error, or the mode in which truncation should occur (undefined bits must be set to zero). These are the effects of setting each bit:

a If bit 0 is set, the result is one byte.
If bit 1 is set, the result is a word.
If bit 2 is set, the result is a longword.
Only one of these three bits may be set.

b If bit 3 is set, then the number is signed.
See notes below.

c If bit 4 is set, the number is unsigned.
See notes below.

d If bit 5 is set, the reference is PC relative, and the relocated current address (ie the address to be updated by this directive) is to be subtracted before the truncation process.

e If bit 6 is set, runtime relocation is requested (for longwords only). The address of the longword is included in a table generated by the linker which can be used by a runtime loader.

After the <truncation rule> is a sequence of terms for the expression. <op> is a one-byte operator code and can be 2B for "+" or 2D for "−". <id> is a symbol or section name id as defined in the DEFINE directive (2.8). The special <id> code of 0000 refers to the current location counter (ie the address updated by this directive).

The final FB byte terminates the sequence of terms in the expression.

As an example of the use of the signed/unsigned bits, consider a value which must be written out as a word value; the signed/unsigned bits are interpreted as follows:

resulting value

	< FFFF8000	always out of range
FFFF8000	– FFFFFFFF	illegal if 'unsigned' bit is set
00000000	– 00007FFF	always allowed
00008000	– 0000FFFF	illegal if 'signed' bit is set
	> 0000FFFF	out of range

There are some examples of XREF directives in B.2.5 below.

B.2.2.8 DEFINE Syntax: FB 10 <id> <symbol>
FB 10 <id> <section name>

This directive is used in conjunction with XDEF, XREF, SECTION and COMMON.

The directive defines that the <symbol> or <section name> may be referenced by the 2-byte <id> in XREF directives. A <section name> has the same syntax as a <symbol>.

Note that positive nonzero <id> values refer to symbols and negative <id> values refer to section names.

This directive must appear before the <id> value is used in any other directive.

If two <id> values are used to refer to the same symbol, or if one <id> value is reassigned to another symbol the effects are undefined at present.

B.2.2.9 COMMON Syntax: FB 12 <id>
This directive is identical to the SECTION directive except that it informs the linker that the section is to be a common section so that references to this section in **different** object modules refer to the same memory location.

Within the same object module multiple additions to the same section will be appended together as for an ordinary section.

When different modules create common sections of differing size, the linker should create a section equal in size to the largest one.

B.2.2.10 END Syntax: FB 13

This directive marks the end of the current object module. If the file contains only one module, then this will appear at the end of file.

B.2.3 Directive ordering

B.2.3.1 Mandatory Rules

Within a relocatable object file the following rules should be applied to the ordering of the directives within an object module:

a) A SECTION directive (or ORG or COMMON) must appear before any data bytes in the module.

b) A symbol or section's <id> must be defined in a DEFINE directive before it is used in any other directive.

The ordering of other directives is at the discretion of the authors of compilers or relocatable assemblers, though it will normally be dictated by the source code.

B.2.3.2 BNF definition of a relocatable object file

This BNF uses { } to mean 0 or more repetitions of an item.

<relocatable object file> = <module> { <module> }

<module> = SOURCE { <chunk> } END

<chunk> = <header> <body>

<header> = { <header command> } <section command>

<header command> = COMMAND | XDEF | DEFINE

<section command> = SECTION | ORG | COMMON

<body> = { <data byte> | <body command> }

<body command> = OFFSET | XDEF | XREF | DEFINE | COMMENT

B.2.4 Library format

B.2.4.1 Use of libraries in the QL Linker

A library is a relocatable object file as described above, but it will normally contain more than one module. Note that a library can be created by appending smaller libraries or object files.

When the linker processes a LIBRARY command it checks each module to see if it resolves any external references. If so, that module will be included in the link.

The linker also has a facility to extract a specific module from a library, using the module name in the source directive.

B.2.5 Example

The object module format will be illustrated with the aid of this example assembler source module: the file name is "MDV1_EXAMPLE_ASM".

The Program is shown in Fig 1.

```
                        TITLE       Illustrate the object module format
                        XDEF        TABLE1,THISROUTINE
                        XREF        FINALTAB
                        XREF        SEARCHTABLE
                        XREF        THATROUTINE, THEOTHER
                        SECTION     CODE
                                    :
                                    :
                                    :
001234                  THISROUTINE:
001234  317Cxxxx0008    MOVE.W      #FINALTAB-TABLE2,8(A0)
00123A  41FAxxxx        LEA         TABLE1(PC),A0
00123E  4EBAxxxx        JSR         SEARCHTABLE
                                    :
                                    :
                                    :
                        SECTION     DATA-TABLES
                                    :
                                    :
                                    :
               *note that this assembler interprets "*" as the address
               *at the start of the current line, not the current pc
000072 xxxxxxxxxxxx TABLE1: DC.L THISROUTINE-*,THATROUTINE-*,THEOTHER-*
                                    :
                                    :
                                    :
0000C8  000102030405 TABLE2: DC.B    0,1,2,3,4,5,6,7,8,9
    END
```

Fig 1.

The generated object module would then look something like this (in file "MDV1-EXAMPLE-REL"):

```
FB 01 10 45 58 41 4D 50 4C 45 20 32 38 2F 30 39 2F
38 34
SOURCE    E X A M P L E    2 8/0 9/8 4

FB 02 23 49 6C 6C 75 ......
COMMENT Illustrate.....

FB 10 FFFF 04 43 4F 44 45
DEFINE -1    CODE

FB 10 FFFE 0B 44 41 54 41 5F 54 41 42 4C 45 53
DEFINE -2    DATA-TABLES

FB 06 06 54 41 42 4C 45 31 00000072 FFFE
XDEF       TABLE1            DATA-TABLES

FB 06 0B 54 48 49 53 52 4F 55 54 49 4E 45
00001234 FFFF
XDEF       THISROUTINE               CODE

FB 10 0001 08 46 49 4E 41 4C 54 41 42
DEFINE     +1      FINALTAB

FB 10 0002 0B 53 45 41 52 43 48 54 41 42 4C 45
DEFINE     +2      SEARCHTABLE

FB 10 0003 0B 54 48 41 54 52 4F 55 54 49 4E 45
DEFINE     +3      THATROUTINE

FB 10 0004 08 54 48 45 4F 54 48 45 52
DEFINE     +4      THEOTHER

FB 02 FFFF
SECTION CODE

...
```

```
317C FB 07 FFFFFF38 02 2B 0001       2D
FFFE      FB 0008
MOVE XREF -C8 | + FINALTAB - DATA-TABLES
                   Rules: word

41FA FB 07 00000072 2A 2B FFFE FB
LEA XREF           | + DATA-TABLES
                   Rules: PC-rel, word, signed

4ECA FB 07 00000000 2A 2B 0002 FB
JSR XREF           | + SEARCHTABLE
                   Rules: PC-rel, word, signed
...

FB 02 FFFE
SECTION DATA-TABLES

...

FB 07 000011C2 04 2B FFFF 2D FFFE FB
XREF 1234-0072 | + CODE — DATA-TABLES
Rules: long

FB 07 FFFFFF8E 04 2B 0003      2D FFFE FB
XREF        -0072   | + THATROUTINE     -
DATA-TABLES
Rules: long

FB 07 FFFFFF8E 04 2B 0004      2D FFFE FB
XREF        -0072    | THEOTHER       +
DATA-TABLES
Rules: long

...

FB 13
END
```

B.3 Program file

The program file created by the linker will be a binary file which can be executed directly by the operating system. The linker will place the following information into the type dependent information section of the file header:

- **The DATA requirement**
  The DATA requirement is the amount of data space in bytes required for the program to run. The system allocates this data space when the program is loaded.

  The value is a longword occupying the first four bytes of the type dependent information field. The value is entered by the DATA command.

- **The OFFSET value**
  This value represents the start address of the file. Normally this value is zero unless the OFFSET command has been used in the linker. A program with a non zero OFFSET in the header should not be run directly.

  The value is a longword occupying the final four bytes of the type dependent information field.

B.4 The listing file

The listing file consists of a series of reports to indicate what the linker has done with the program file. The following reports are generated.

- **Command line and control file information**
  This report indicates the command line used to perform the link and a listing of the control file (if one was used). Any error messages from processing of the control file are also placed in the report.

- **Object module header information**
  This report indicates which commands were used for input of modules and the module names read in by the command. Any error messages produced while reading the module files are also printed here.

- **Load Map**
  This report generated after pass 1 indicates where the linker has placed everything. The load map is produced in increasing address order with the following format:

  - For each section a line in the following form:
    1 The section type (ABSOLUTE, SECTION, COMMON)
    2 The section start address
    3 The section end address
    4 The section name

  - For each subsection (contribution from a file) a line of the following form:
    1 The start address of the subsection
    2 The end address of the subsection
    3 The module name

  - For each entry point in a relocatable or common subsection a line of the following form (in increasing address order)
    1 The entry point address
    2 The entry point name

  The load map is then followed by three lists of the following form:
  1 Absolute symbols in address order
  2 User defined symbols in defined order
  3 Undefined symbols in alphabetical order

- **Symbol table listing**
  The linker produces a symbol table listing of all global symbols in the link in alphabetical order. For each symbol a line is printed containing the following information:

  1 The value of the symbol (or ???????? for undefined symbols)
  2 The symbol name
  3 The section name the symbol is defined in, or Absolute (defined or undefined).
  4 The module name (unless defined or undefined).

If the –CRF option is used on the command line then if a symbol is referenced in other modules the symbol information is followed by one or more lines of module names which reference the symbol. This cross reference information is followed by a blank line before the next symbol table entry.

B.5 The debug file

The debug file is a text file containing a symbol table listing in fixed format for use by a symbolic debugger. The symbol table is sorted alphabetically with one line of information for each symbol in the following format:

1. An 8 digit hex number representing the symbol's value
2. A single space
3. The symbol type letter which will be one of the following:

   A Absolute
   C Common
   D User Defined
   R Relocated
   U Undefined

4. A single space
5. The symbol name
6. A newline character

Appendix C – Glossary

The following terms are applied throughout in this manual.

absolute section
A section of code that starts at a specific address in the program file. Absolute sections are placed in the program file at their start addresses relative to the start of the file (or the OFFSET value if supplied). Absolute sections are used for position dependent code (i.e. not directly runnable by the operating system) and are very rarely encountered.

common sections
A common section is a section where all contributions to the section from different files refer to the same memory locations. This type of section is used in the implementation of FORTRAN type common blocks.

default parameter
A parameter generated in the absence of an explicit parameter. Used in the command line handling to generate file names for the program listing and debug files.

module
Binary data in Sinclair relocatable object module format. Generated by compilers and assemblers and used as inputs to the linker to generate a program file. The name of the module is usually part of a filename but may be any name with the same syntax as a symbol.

positional parameter
Any parameter whose meaning is determined from its position within a command line.

relocatable binary file
A file containing one or more relocatable binary modules (see module).

relocatable section
see section

section
A section is a portion of data or code that logically belongs together. Contributions may be made to a section from one or more relocatable object modules. A section may also be overlayed (see common section).

A section name has the same syntax as a symbol.

symbol
A symbol is a name of up to 32 characters with the following syntax:

<symbol name> = <letter> { <symchar> }

<symchar> = <letter> | <digit> | _ | $ | .

<letter> = A | B | ... | Y | Z

<digit> = 0 | 1 | ... | 8 | 9

QL·Screen Editor

Contents

1. Introduction

The screen editor ED may be used to create a new file or to alter an existing one. The text is displayed on the screen, and can be scrolled vertically or horizontally as required. The size of the program is about 20K bytes and it requires a minimum workspace of 8K bytes.

The editor is invoked using EXEC or EXEC_W as follows

```
EXEC_W mdv1_ed
```

The difference between invoking a program with EXEC or EXEC_W is as follows. Using EXEC_W means that the editor is loaded and SuperBASIC waits until the editing is complete. Anything typed while the editor is running is directed to the editor. When the editor finishes, keyboard input is directed at SuperBASIC once more.

Using EXEC is slightly more complicated but is more flexible. In this case the editor is loaded into rnemory and is started, but SuperBASIC carries on running. Anything typed at the keyboard is directed to SuperBASIC unless the current window is changed. This is performed by typing CTRL-C (pressing the CONTROL key and C together), which switches to another window. If just one copy of ED is running then CTRL-C will switch to the editor window, and characters typed at the keyboard will be directed to the editor. A subsequent CTRL-C switches back to SuperBASIC. When the editor is terminated a CTRL-C will be needed to switch back to SuperBASIC once more. More than one version of the editor can be run concurrently (subject to available memory) if EXEC is used. In this case CTRL-C switches between SuperBASIC and the two versions of the editor in turn.

Once the program is loaded it will ask for a file name which should conform to the standard Qdos file name syntax. No check is made on the name used, but if it is invalid a message will be issued when an attempt is made to write the file out, and a different file name may be specified then if required. All subsequent questions have defaults which are obtained by just pressing ENTER.

The next question asks for the workspace required. ED works by loading the file to be edited into memory and sufficient workspace is needed to hold all the file plus a small overhead. The default is 12K bytes which is sufficient for small files. The amount can be specified as a number or in units of 1024 bytes if the number is terminated by the character K. If you ask for more memory than is available then the question is asked again. The minimum is 8K bytes.

You are next asked if you wish to alter the window used by ED. The default window is normally the same as the window used in the initialisation of ED although this may be altered if required. See Appendix A for details of how to do this. If you type **N** or just press **ENTER** then the default window is used. If you type **Y** then you are given a chance to alter the window. The current window is displayed on the screen and the cursor keys can be used to move the window around. The combination **ALT** and the cursor keys will alter the size of the window although there is a minimum size which may be used. Within this constraint you can specify a window anywhere on the screen, so that you can edit a file and do something else such as run a SuperBASIC program concurrently. When you are satisfied with the position of the window press **ENTER**.

Next, an attempt is made to open the file specified, and if this succeeds then the file is read into storage and the first few lines displayed on the screen. Otherwise a blank screen is provided, ready for the addition of new data. The message "File too big" indicates that more workspace should be specified.

When the editor is running the bottom line of the screen is used as a message area and command line. Any error messages are displayed there, and remain displayed until another editor command is given.

Editor commands fall into two categories – immediate commands and extended commands. Immediate commands are those which are executed immediately, and are specified by a single key or control key combination. Extended commands are typed in onto the command line, and are not executed until the command line is finished. A number of extended commands may be typed on a single command line, and any commands may be grouped together and groups repeated automatically. Most immediate commands have a matching extended version.

Immediate commands use the function keys and cursor keys on the QL in conjunction with the special keys SHIFT, CTRL and ALT. For example, delete line is requested by holding down the CTRL and ALT keys and then pressing the left arrow key. This is described in this document as CTRL-ALT-LEFT. Function keys are described as F1, F2 etc.

The editor attempts to keep the screen up to date, but if a further command is entered while it is attempting to redraw the display, the command is executed at once and the display will be updated later, when there is time. The current line is always displayed first, and is always up to date.

2. Immediate commands

2.1 Cursor control

The cursor is moved one position in either direction by the cursor control keys LEFT, RIGHT, UP and DOWN. If the cursor is on the edge of the screen the text is scrolled to make the rest of the text visible. Vertical scroll is carried out a line at a time, while horizontal scroll is carried out ten characters at a time. The cursor cannot be moved off the top or bottom of the file, or off the left hand edge of the text.

The ALT-RIGHT combination will take the cursor to the right hand edge of the current line, while ALT-LEFT moves it to the left hand edge of the line. The text will be scrolled horizontally if required. In a similar fashion SHIFT-UP places the cursor at the start of the first line on the screen, and SHIFT-DOWN places it at the end of the last line on the screen.

The combinations SHIFT-RIGHT and SHIFT-LEFT take the cursor to the start of the next word or to the space following the previous word respectively. The text will be scrolled vertically or horizontally as required. The TAB key can also be used. If the cursor position is beyond the end of the current line then TAB moves the cursor to the next tab position, which is a multiple of the tab setting (initially 3). If the cursor is over some text then sufficient spaces are inserted to align the cursor with the next tab position, with any characters to the right of the cursor being shuffled to the right.

2.2 Inserting text

Any letter typed will be added to the text in the position indicated by the cursor, unless the line is too long (there is a maximum of 255 characters in a line). Any characters to the right of the text will be shuffled up to make room. If the line exceeds the size of the screen the end of the line will disappear and will be redisplayed when the text is scrolled horizontally. If the cursor has been placed beyond the end of the line, for example by means of the TAB or cursor control keys, then spaces are inserted between the end of the line and any inserted character. Although the QL keyboard generates a different code for SHIFT-SPACE and SHIFT-ENTER these are mapped to normal SPACE and ENTER characters for convenience.

An ENTER key causes the current line to be split at the position indicated by the cursor, and a new line generated. If the cursor is at the end of a line the effect is simply to create a new, empty blank line after the current one. Alternatively CTRL-DOWN may be used to generate a blank line after the current, with no split of the current line taking place. In either case the cursor is placed on the new line at the position indicated by the left margin (initially column one).

A right margin may be set up so that ENTERs are automatically inserted before the preceding word when the length of the line being typed exceeds that margin. In detail, if a character is typed and the cursor is at the end of the line and at the right margin position then an automatic new line is generated. Unless the character typed was a space, the half completed word at the end of the line is moved down to the newly generated line. Initially there is a right margin set up at the right hand edge of the window used by ED. The right margin may be disabled by means of the EX command (see later).

2.3 Deleting text

The CTRL-LEFT key combination deletes the character to the left of the cursor and moves the cursor left one position. If the cursor is at the start of a line then the new line between the current line and the previous is deleted (unless you are on the very first line). The text will be scrolled if required. CTRL-RIGHT deletes the character at the current cursor position without moving the cursor. As with all deletes, characters remaining on the line are shuffled down, and text which was invisible beyond the right hand edge of the screen may now become visible.

The combination SHIFT-CTRL-RIGHT may be used to delete a word or a number of spaces. The action of this depends on the character at the cursor. If this character is a space then all spaces up to the next non-space character on the line are deleted. Otherwise characters are deleted from the cursor, and text shuffled left, until a space is found. The CTRL-ALT-RIGHT command deletes all characters from the cursor to the end of the line. The CTRL-ALT-LEFT command deletes the entire current line.

2.4 Scrolling

Besides the vertical scroll of one line obtained by moving the cursor to the edge of the screen, the text may be scrolled 12 lines vertically by means of the commands ALT-UP and ALT-DOWN. ALT-UP moves to the previous lines, moving the text window up; ALT-DOWN moves the text window down moving to lines further on in the file. The F4 key rewrites the entire screen, which is useful if the screen is altered by another program besides the editor. Remember that you can switch out of the editor window and into some other job by typing CTRL-C at any point, assuming that there is another job with an outstanding input request. SuperBASIC will be available only if you entered the editor using EXEC rather than EXEC_W. If there is enough room in memory you can run two versions of ED at the same time if you wish.

2.5 Repeating commands

The editor remembers any extended command line typed, and this set of extended commands may be executed again at any time by simply pressing F2. Thus a search command could be set up as the extended command, and executed in the normal way. If the first occurrence found is not the one required, typing F2 will cause the search to be executed again. As most immediate commands have an extended version, complex sets of editing commands can be set up and executed many times. Note that if the extended command line contains repetition counts then the relevant commands in the group will be executed many times each time the F2 key is pressed.

3. Extended commands

Extended command mode is entered by pressing the F3 key. Subsequent input will appear on the command line at the bottom of the screen. Mistakes may be corrected by means of CTRL-LEFT and CTRL-RIGHT in the normal way, while LEFT and RIGHT move the cursor over the command line. The command line is terminated by pressing ENTER. After the extended command has been executed the editor reverts to immediate mode. Note that many extended commands can be given on a single command line, but the maximum length of the command line is 255 characters. An empty command line is allowed, so just typing ENTER after typing F3 will return to immediate mode.

Extended commands consist of one or two letters, with upper and lower case regarded as the same. Multiple commands on the same command line are separated from each other by a semicolon. Commands are sometimes followed by an argument, such as a number or a string. A string is a sequence of letters introduced and terminated by a delimiter, which is any character besides letters, numbers, space, semicolon or brackets. Thus valid strings might be

```
/happy/  !23feet!:Hello!  :"1/2"
```

Most immediate commands have a corresponding extended version. See the table of commands for full details (section 4).

3.1 Program control

The command X causes the editor to exit. The text held in storage is written out to file, and the editor then terminates. The editor may fail to write the file out either because the file name specified when editing started was invalid, or because the Microdrive becomes full. In either case the editor remains running, and a new destination should be specified by means of the SA command described below. Alternatively the Q command terminates immediately without writing the buffer; confirmation is requested in this case if any changes have been made to the file. A further command allows a 'snapshot' copy of the file to be made without coming out of ED. This is the SA command. SA saves the text to a named file or, in the absence of a named file, to the current file. For example:

```
*SA/mdv2_savedtext/
```
or
```
*SA
```

This command is particularly useful in areas subject to power failure or surge. It should be noted that **SA** followed by **Q** is equivalent to the **X** command. Any alterations made between the **SA** and the **Q** will cause ED to request confirmation again; if no alterations have been made the program will be quitted immediately with the file saved in that state. **SA** is also useful because it allows the user to specify a file name other than the current one. It is therefore possible to make copies at different stages and place them in different files.

The **SA** command is also useful in conjunction with the **R** command. Typing **R** followed by a file name causes the editor to be re-entered editing the new file. The old file will be lost when this happens, so confirmation is requested (as with the **Q** command) if any changes to the current file have been made. The normal action is therefore to save the current file with **SA**, and then start editing a new file with **R**. This saves having to load the editor into memory again, and means that once the editor is loaded the Microdrive containing it can be replaced by another.

The **U** command 'undoes' any alterations made to the current line if possible. When the cursor is moved from one line to another, the editor takes a copy of the new line before making any changes to it. The **U** command causes the copy to be restored. However the old copy is discarded and a new one made in a number of circumstances. These are when the cursor is moved off the current line, or when scrolling in a horizontal or vertical direction is performed, or when any extended command which alters the current line is used. Thus **U** will not 'undo' a delete line or insert line command, because the cursor has been moved off the current line.

The **SH** command shows the current state of the editor. Information such as the value of tab stops, current margins, block marks and the name of the file being edited is displayed. Tabs are initially set at every three columns; this can be changed by the command **ST**, followed by a number n, which sets tabs at every n columns. The left margin and right margin can be set by **SL** and **SR** commands, again followed by a number indicating the column position. The left margin should not be set beyond the width of the screen. The `EX` command may be used to extend margins; once this command is given no account will be taken of the right margin on the current line. Once the cursor is moved off the current line, margins are enabled once more.

3.2 Block control

A block of text can be identified by means of the `BS` (block start) and `BE` (block end) commands. The cursor should be moved to the first line required in a block, and the `BS` command given. The cursor can then be moved to the last line wanted in the block, by cursor control commands or in any other way, such as searching. The `BE` command is then used to mark the end of the block. Note, however, that if any change is made to the text the block start and block end become undefined once more. The start of the block must be on the same line, or a line previous to, the line which marks the end of the block. A block always contains all of the line(s) within it.

Once a block has been identified, a copy of it may be moved into another part of the file by means of the `IB` (insert block) command. The previously identified block is replicated immediately after the current line. Alternatively a block may be deleted by means of the `DB` command, after which the block start and end values are undefined. It is not possible to insert a block within itself.

Block marks may also be used to remember a place in a file. The `SB` (show block) command resets the screen window on the file so that the first line in the block is at the top of the screen.

A block may also be written to a file by means of the `WB` command. The command is followed by a string which represents a file name. The file is created, possibly destroying the previous contents, and the buffer written to it. A file may be inserted by the `IF` command. The file name given as the argument string is read into storage immediately following the current line.

3.3 Movement

The command T moves the screen to the top of the file, so that the first line in the file is the first line on the screen. The B command moves the screen to the bottom of the file, so that the last line in the file is the bottom line on the screen if possible.

The commands N and P move the cursor to the start of the next line and previous line respectively. The commands CL and CR move the cursor one place to the left or one place to the right, while CE places the cursor at the end of the current line, and CS places it at the start.

It is common for programs such as compilers and assemblers to give line numbers to indicate where an error has been detected. For this reason the command M is provided, which is followed by a number representing the line number which is to be located. The cursor will be placed on the line number in question. Thus M1 is the same as the T command. If the line number specified is too large the cursor will be placed at the end of the file.

3.4 Searching and Exchanging

Alternatively the screen window may be moved to a particular context. The command F is followed by a string which represents the text to be located. The search starts at one place beyond the current cursor position and continues forwards through the file. If found, the cursor is placed at the start of the located string. To search backwards through the text use the command BF (backwards find) in the same way as F. BF will find the last occurrence of the string before the current cursor position. To find the earliest occurrence use T followed by F; to find the last, use B followed by BF. The string after F and BF can be omitted; in this case the string specified in the last F, BF or E command is used. Thus

```
*F /wombat/
*BF
```

will search for 'wombat' in a forwards direction and then in a reverse direction.

The E (exchange) command takes a string followed by further text and a further delimiter character, and causes the first string to be exchanged to the last. So for example

```
E /wombat/zebra/
```

would cause the letters 'wombat' to be changed to 'zebra'. The editor will start searching for the first string at the current cursor position, and continues through the file. After the exchange is done the cursor is moved to after the exchanged text. An empty string is allowed as the search string, specified by two delimiters with nothing between them. In this case the second string is inserted at the current cursor position. No account is taken of margin settings while exchanging text.

A variant on the E command is the EQ command. This queries the user whether the exchange should take place before it happens. If the response is N then the cursor is moved past the search string. If the response is Y or ENTER then the change takes place; any other response (except F2) will cause the command to be abandoned. This command is normally only useful in repeated groups; a response such as Q can be used to exit from an infinite repetition.

All of these commands normally perform the search making a distinction between upper and lower case. The command UC may be given which causes all subsequent searches to be made with cases equated. Once this command has been given then the search string 'wombat' will match 'Wombat', 'WOMBAT', 'WoMbAt' and so on. The distinction can be enabled again by the command LC.

3.5 Altering text

The E command cannot be used to insert a new line into the text, but the I and A commands may be used instead. The I command is followed by a string which is inserted as a complete line before the current line. The A command is also followed by a string, which is inserted after the current line. It is possible to add control characters into a file in this way.

The S command splits the current line at the cursor position, and acts just as though an ENTER had been typed in immediate mode. The J command joins the next line on to the end of the current one.

The D command deletes the current line in the same way as the CTRL-ALT-LEFT command in immediate mode, while the DC command deletes the character at the cursor in the same way as CTRL-RIGHT.

3.6 Repeating commands

Any command may be repeated by preceding it with a number. For example,

```
4E/slithy/brillig/
```

will change the next four occurrences of 'slithy' to 'brillig'. The screen is verified after each command. The RP (repeat) command can be used to repeat a command until an error is reported, such as reaching the end of the file. For example,

```
RP E/slithy/brillig/
```

will change all occurrences of 'slithy' to 'brillig'.

Commands may be grouped together with brackets and these command groups executed repeatedly. Command groups may contain further nested command groups. For example,

```
RP (/f/toves/;3(IB;N))
```

will insert three copies of the current block whenever the string 'toves' is located.

Note that some commands are possible, but silly. For example,

```
RP SR 60
```

will set the right margin to 60 ad infinitum. However, any sequence of extended commands, and particularly repeated ones, can be interrupted by typing any character while they are taking place. Command sequences are also abandoned if any error occurs.

4. Command list

4.1 Immediate commands

F2	Repeat last extended command
F3	Enter extended mode
F4	Redraw screen
LEFT	Move cursor left
SHIFT-LEFT	Move cursor to previous word
ALT-LEFT	Move cursor to start of line
CTRL-LEFT	Delete left one character
CTRL-ALT-LEFT	Delete line
RIGHT	Move cursor right
SHIFT-RIGHT	Move cursor to start of next word
ALT-RIGHT	Move cursor to end of line
CTRL-RIGHT	Delete right one character
CTRL-ALT-RIGHT	Delete to end of line
SHIFT-CTRL-RIGHT	Delete word to right
UP	Move cursor up
SHIFT-UP	Cursor to top of screen
ALT-UP	Scroll up
DOWN	Move cursor down
SHIFT-DOWN	Cursor to bottom of screen
ALT-DOWN	Scroll down
CTRL-DOWN	Insert blank line

4.2 Extended commands

/s/ indicates a string, /s/t/ indicates two exchange strings and n indicates a number.

A/s/	Insert line after current
B	Move to bottom of file
BE	Block end at cursor
BF	Backwards find
BS	Block start at cursor
CE	Move cursor to end of line
CL	Move cursor one position left
CR	Move cursor one position right

CS	Move cursor to start of line
D	Delete current line
DB	Delete block
DC	Delete character at cursor
E/s/t/	Exchange s into t
EQ/s/t/	Exchange but query first
EX	Extend right margin
F/s/	Find string s
I/s/	Insert line before current
IB	Insert copy of block
IF/s/	Insert file s
J	Join current line with next
LC	Distinguish between upper and lower case in searches
M n	Move to line n
N	Move cursor to start of next line
P	Move cursor to start of previous line
Q	Quit without saving text
R/s/	Re-enter editor with file s
RP	Repeat until error
S	Split line at cursor
SA/s/	Save text to file
SB	Show block on screen
SH	Show information
SL n	Set left margin
SR n	Set right margin
ST n	Set tab distance
T	Move to top of file
U	Undo changes on current line
UC	Equate U/C and l/c in searches
WB/s/	Write block to file s
X	Exit, writing text back.

Appendix A:
Changing the default window

The window to be used can be altered as part of the initialisation sequence. If this option is not required then the default window is used. This is initially the same as the window used during the start of the program, but if required the default window may be altered permanently by patching the programs. This is useful where a certain window size and position is always required and means that the window does not have to be positioned correctly each time the program is run.

The program INSTALL is supplied on the distribution Microdrive for this purpose. It is run by the command

```
LRUN mdv1_install
```

The program starts by asking whether the default window is to be set up for TV or monitor mode. The minimum window size is greater in TV mode because the characters used are larger. You should answer T if you are setting the default for use with TV mode and M if you are setting it for use with monitor mode. Note that the current mode in use is of no consequence.

The standard window will appear on the screen and can be moved by means of the cursor keys and altered in size by means of ALT cursor keys. This is similar to the mechanism used when altering the window during normal program initialisation. Once the window is in the right place and of the desired size, press ENTER.

The program now asks for the name of the file which is to be modified. If you wished to alter the editor then the file would probably be something like 'mdv1_ed'. The next item requested is the name of the program. When a new job is running on the QL, it has a name associated with it. This can be inspected by suitable utilities. The name is six characters long, and whatever is typed here is used as the name and forced to the correct length. The name is of little importance except for job identification.

The INSTALL program will then modify the file specified. INSTALL can be run as many times as you like to alter the default window of the editor.