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

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

– **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 they 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 fife. 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            TABLE 1, THIS ROUTINE
                                        XREF            FINAL TAB
                                        XREF            SEARCH TABLE
                                        XREF            THAT ROUTINE, THE OTHER
                                        SECTION         CODE
                                                        :
                                                        :
                                                        :
001234                                  THIS ROUTINE:
001234  317Cxxxx0008                   MOVE.W          # FINAL TAB - TABLE 2,8(A0)
00123A  41FAxxxx                        LEA             TABLE 1(PC),A0
00123E  4EBAxxxx                        JSR             SEARCH TABLE
                                                        :
                                                        :
                                                        :
                                        SECTION         DATA_TABLES
                                                        :
                                                        :
                                                        :
        *note that this assembler interprets "*" as the address
        *at the start of the current line, not the current pc
000072  xxxxxxxxxxxx TABLE 1 : DC.L THIS ROUTINE-*,THAT ROUTINE-*,THE OTHER-*
                                                        :
                                                        :
                                                        :
0000C8  000102030405 TABLE 2: 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          EXAMPLE         28/09/84

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

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

FB 10 FF FE 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 00 00 00 72 FF FE
XDEF            TABLE 1   DATA_TABLES

FB 06 0B 54 48 49 53 52 4F 55 54 49 4E 45
00 00 12 34 FF FF
XDEF            THIS ROUTINE                CODE

FB 10 00 01 08 46 49 4E 41 4C 54 41 42
DEFINE        +1        FINAL TAB

FB 10 00 02 0B 53 45 41 52 43 48 54 41 42 4C 45
DEFINE        +2        SEARCH TABLE

FB 10 00 03 0B 54 48 41 54 52 4F 55 54 49 4E 45
DEFINE        +3        THAT ROUTINE

FB 10 00 04 08 54 48 45 4F 54 58 45 52
DEFINE        +4        THE OTHER

FB 02 FF FF
SECTION CODE

. . .
```

```
31 7C FB 07 FF FF FF 38 02 2B 00 01       2D
FF FE          FB 00 08
MOVE XREF -C8 | + FINAL TAB - DATA-TABLES
                    Rules: word

41 FA FB 07 00 00 00 72 2A 2B FF FE FB
LEA XREF     |  + DATA-TABLES
                    Rules:   PC - rel, word, signed

4E CA FB 07 00 00 00 00 2A 2B 00 02 FB
JSR  XREF    |  +SEARCH TABLE
                    Rules: PC-rel, word, signed
. . .

FB 02 FF FE
SECTION DATA-TABLES

. . .

FB 07 00 00 11 C2 04 2B FF FF 2D FF FE FB
XREF 1234-00 72  |  + CODE–DATA-TABLES
Rules: long

FB 07 FF FF FF 8E 04 26 00 03        2D FF FE FB
XREF              -00 72  |  + THAT ROUTINE   - DATA-TABLES
Rules: long

FB 07 FF FF FF 8E 04 2B 00 04        2D FF FE FB
XREF              -0072   |  THE OTHER          + 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