IAR Linker and Library Tools

Reference Guide

Version 4.53

EDITION NOTICE
Twelfth edition: April 2002

Part number: XLINK-453K

The IAR Linker and Library Tools Reference Guide replaces all versions of the IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide.

Contents

Tables

Preface

Welcome to the IAR Linker and Library Tools Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR linker and library tools to best suit your application requirements.

Who should read this guide

This guide provides reference information about the IAR XLINK Linker™ version 4.53, the IAR XAR Library Builder™, and the IAR XLIB Librarian™. You should read it if you plan to use the IAR Systems tools for linking your applications and need to get detailed reference information on how to use the IAR linker and library tools. In addition, you should have working knowledge of the following:

- The architecture and instruction set of your target microcontroller. Refer to the chip manufacturer's documentation.
- Your host operating system.

For information about programming with the IAR Compiler, refer to the *IAR Compiler Reference Guide*.

For information about programming with the IAR Assembler, refer to the *IAR Assembler Reference Guide*.

How to use this guide

When you first begin using IAR linker and library tools, you should read the *Introduction to the IAR XLINK Linker* and *Introduction to the IAR library tools* chapters in this reference guide.

If you are an intermediate or advanced user, you can focus more on the reference chapters that follow the introductions.

If you are new to using the IAR toolkit, we recommend that you first read the initial chapters of the *IAR Embedded Workbench™ User Guide*, where you will find information about installing the IAR Systems development tools, product overviews, and tutorials that will help you get started. The *IAR Embedded Workbench™ User Guide* also contains complete reference information about the IAR Embedded Workbench and the IAR C-SPY™ Debugger.

What this guide contains

Below is a brief outline and summary of the chapters in this guide.

Part 1: The IAR XLINK Linker

- *Introduction to the IAR XLINK Linker* describes the IAR XLINK Linker, and gives examples of how it can be used. It also explains the XLINK listing format.
- *XLINK options* describes how to set the XLINK options, gives an alphabetical summary of the options, and provides detailed information about each option.
- *XLINK output formats* summarizes the output formats available from XLINK.
- *XLINK environment variables* gives reference information about the IAR XLINK Linker environment variables.
- *XLINK diagnostics* describes the error and warning messages produced by the IAR XLINK Linker.

Part 2: The IAR Library Tools

- *Introduction to the IAR library tools* describes the IAR library tools—IAR XAR Library Builder and IAR XLIB Librarian—which are designed to allow you to create and maintain relocatable libraries of routines.
- *XAR* describes how to use XAR and gives a summary of the XAR command line options.
- *XAR diagnostics* describes the error and warning messages produced by the IAR XAR Library Builder.
- *XLIB options* gives a summary of the XLIB commands, and complete reference information about each command. It also gives reference information about the IAR XLIB Librarian environment variables.
- *XLIB diagnostics* describes the error and warning messages produced by the IAR XLIB Librarian.

Document conventions

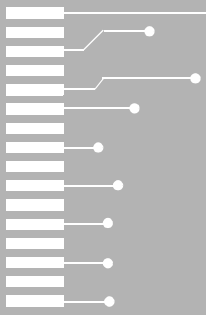
This guide uses the following typographic conventions:

Style	Used for
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.

Table 1: Typographic conventions used in this guide

Style	Used for
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	Cross-references to another part of this guide, or to another guide.
	Identifies instructions specific to the versions of the IAR Systems tools for the IAR Embedded Workbench interface.

Table 1: Typographic conventions used in this guide (Continued)



Part 1: The IAR XLINK Linker

This part of the IAR Linker and Library Tools Reference Guide contains the following chapters:

- Introduction to the IAR XLINK Linker
- XLINK options
- XLINK output formats
- XLINK environment variables
- XLINK diagnostics.

Introduction to the IAR XLINK Linker

The following chapter describes the IAR XLINK Linker™, and gives examples of how it can be used.

Note: The IAR XLINK Linker is a general tool. Therefore, some of the options and segment types described in the following chapters may not be relevant for your product.

Key features

The IAR XLINK Linker converts one or more relocatable object files produced by the IAR Systems Assembler or Compiler to machine code for a specified target processor. It supports a wide range of industry-standard loader formats, in addition to the IAR Systems debug format used by the IAR C-SPY Debugger.

The IAR XLINK Linker supports user libraries, and will load only those modules that are actually needed by the program you are linking.

The final output produced by the IAR XLINK Linker is an absolute, target-executable object file that can be programmed into an EPROM, downloaded to a hardware emulator, or run directly on the host using the IAR C-SPY Debugger.

The IAR XLINK Linker offers the following important features:

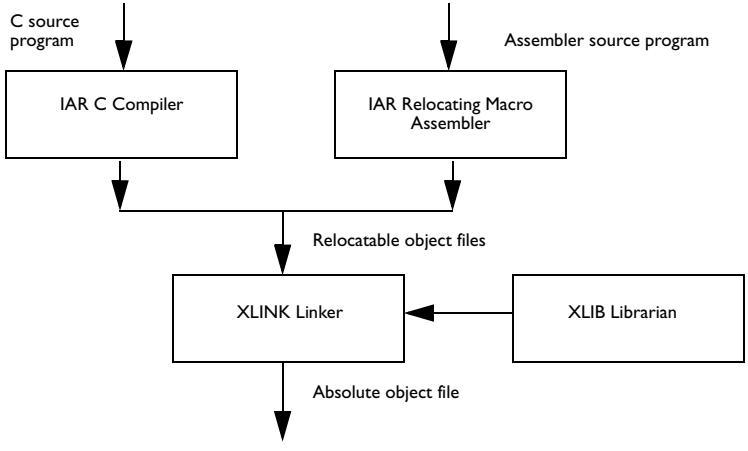
- Unlimited number of input files.
- Searches user-defined library files and loads only those modules needed by the application.
- Symbols may be up to 255 characters long with all characters being significant. Both uppercase and lowercase may be used.
- Global symbols can be defined at link time.
- Flexible segment commands allow full control of the locations of relocatable code and data in memory.
- Support for over 30 output formats.

The linking process

The IAR XLINK Linker is a powerful, flexible software tool for use in the development of embedded-controller applications. XLINK reads one or more relocatable object files produced by the IAR Systems Assembler or Compiler and produces absolute, machine-code programs as output.

It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C/Embedded C++, or mixed C/Embedded C++ and assembler programs.

The following diagram illustrates the linking process:



OBJECT FORMAT

The object files produced by the IAR Systems Assembler and Compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. An application can be made up of any number of UBROF relocatable files, in any combination of assembler and C/Embedded C++ programs.

XLINK FUNCTIONS

The IAR XLINK Linker performs three distinct functions when you link a program:

- It loads modules containing executable code or data from the input file(s).
- It links the various modules together by resolving all global (i.e. non-local, program-wide) symbols that could not be resolved by the assembler or compiler.

- It loads modules needed by the program from user-defined or IAR-supplied libraries.
- It locates each segment of code or data at a user-specified address.

LIBRARIES

When the IAR XLINK Linker reads a library file (which can contain multiple C/Embedded C++ or assembler modules) it will only load those modules which are actually needed by the program you are linking. The IAR XLIB Librarian is used for managing these library files.

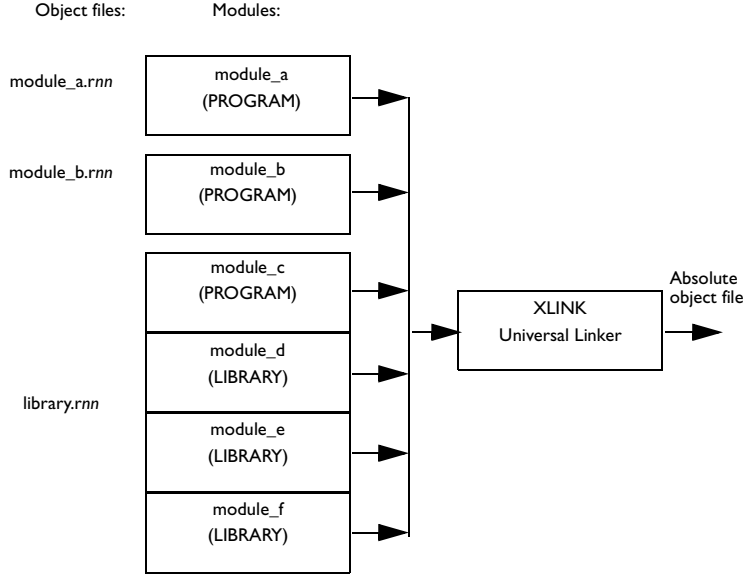
OUTPUT FORMAT

The final output produced by the IAR XLINK Linker is an absolute, executable object file that can be put into an EPROM, downloaded to a hardware emulator, or executed on the PC using the IAR C-SPY™ Debugger.

Note: The default output format in the IAR Embedded Workbench is `DEBUG`.

Input files and modules

The following diagram shows how the IAR XLINK Linker processes input files and load modules for a typical assembler or C/Embedded C++ program:



The main program has been assembled from two source files, `module_a.snn` and `module_b.snn`, to produce two relocatable files. Each of these files consists of a single module `module_a` and `module_b`. By default, the assembler assigns the `PROGRAM` attribute to both `module_a` and `module_b`. This means that they will always be loaded and linked whenever the files they are contained in are processed by the IAR XLINK Linker.

The code and data from a single C/Embedded C++ source file ends up as a single module in the file produced by the compiler. In other words, there is a one-to-one relationship between C/Embedded C++ source files and C/Embedded C++ modules. By default, the compiler gives this module the same name as the original C/Embedded C++ source file. Libraries of multiple C/Embedded C++ modules can only be created using the IAR XAR Library Builder™ or the IAR XLIB Librarian™.

Assembler programs can be constructed so that a single source file contains multiple modules, each of which can be a program module or a library module.

LIBRARIES

In the previous diagram, the file `library.rnn` consists of multiple modules, each of which could have been produced by the assembler or the compiler.

The module `module_c`, which has the `PROGRAM` attribute will *always* be loaded whenever the `library.rnn` file is listed among the input files for the linker. In the run-time libraries, the startup module `cstartup` (which is a required module in all C/Embedded C++ programs) has the `PROGRAM` attribute so that it will always get included when you link a C/Embedded C++ project.

The other modules in the `library.rnn` file have the `LIBRARY` attribute. Library modules are only loaded if they contain an entry (a function, variable, or other symbol declared as `PUBLIC`) that is referenced in some way by another module that is loaded. This way, the IAR XLINK Linker only gets the modules from the library file that it needs to build the program. For example, if the entries in `module_e` are not referenced by any loaded module, `module_e` will not be loaded.

This works as follows:

If `module_a` makes a reference to an external symbol, the IAR XLINK Linker will search the other input files for a module containing that symbol as a `PUBLIC` entry; in other words a module where the entry itself is located. If it finds the symbol declared as `PUBLIC` in `module_c`, it will then load that module (if it has not already been loaded). This procedure is iterative, so if `module_c` makes a reference to an external symbol the same thing happens.

It is important to understand that a library file is just like any other relocatable object file. There is really no distinct type of file called a library (modules have a LIBRARY or PROGRAM attribute). What makes a file a library is what it contains and how it is used. Put simply, a library is an *rnn* file that contains a group of related, often-used modules, most of which have a LIBRARY attribute so that they can be loaded on a demand-only basis.

Creating libraries

You can create your own libraries, or extend existing libraries, using C/Embedded C++ or assembler modules. The compiler option --library_module (-b for some IAR products) can be used for making a C/Embedded C++ module have a LIBRARY attribute instead of the default PROGRAM attribute. In assembler programs, the MODULE directive is used for giving a module the LIBRARY attribute, and the NAME directive is used for giving a module the PROGRAM attribute.

The IAR XLIB Librarian is used for creating and managing libraries. Among other tasks, it can be used for altering the attribute (PROGRAM/LIBRARY) of any other module after it has been compiled or assembled.

SEGMENTS

Once the IAR XLINK Linker has identified the modules to be loaded for a program, one of its most important functions is to assign load addresses to the various code and data segments that are being used by the program.

In assembly language programs the programmer is responsible for declaring and naming relocatable segments and determining how they are used. In C/Embedded C++ programs the compiler creates and uses a set of predefined code and data segments, and the programmer has only limited control over segment naming and usage.

Each module contains a number of segment parts. Each segment part belongs to a segment, and contains either bytes of code or data, or reserves space in RAM. Using the XLINK segment control command line options (-Z, -P, and -b), you can cause load addresses to be assigned to segments and segment parts.

After module linking is completed, XLINK removes the segment parts that were not required. It accomplishes this by first including all ROOT segment parts in loaded modules, and then adding enough other segment parts to satisfy all dependencies. Dependencies are either references to external symbols defined in other modules or segment part references within a module. The ROOT segment parts normally consists of the root of the C run-time boot process and any interrupt vector elements.

Compilers and assemblers that produce UBROF 7 or later can put individual functions and variables into separate segment parts, and can represent all dependencies between segment parts in the object file. This enables XLINK to exclude functions and variables that are not required in the build process.

Segment control

The following options control the allocation of segments.

`-K`*`segs=inc,count`*	Duplicate code.
`-P`*`pack_def`*	Define packed segments.
`-Z`*`seg_def`*	Define segments.
`-b`*`bank_def`*	Define banked segments.
`-M`*`range_def`*	Map logical addresses to physical addresses.

For detailed information about the options, see the chapter *XLINK options*, page 17.

Segment placement using `-Z` and `-P` is performed one placement command at a time, taking previous placement commands into account. As each placement command is processed, any part of the ranges given for that placement command that is already in use is removed from the considered ranges. Memory ranges can be in use either by segments placed by earlier segment placement commands, by segment duplication, or by objects placed at absolute addresses in the input fields.

For example, if there are two data segments (`Z1`, `Z2`) that must be placed in the zero page (`0-FF`) and three (`A1`, `A2`, `A3`) that can be placed anywhere in available RAM, they can be placed like this:

```
-Z(DATA)Z1,Z2=0-FF
-Z(DATA)A1,A2,A3=0-1FFF
```

This will place `Z1` and `Z2` from 0 and up, giving an error if they do not fit into the range given, and then place `A1`, `A2`, and `A3` from the first address not used by `Z1` and `Z2`.

The `-P` option differs from `-Z` in that it does not necessarily place the segments (or segment parts) sequentially. See page 31 for more information about the `-P` option. With `-P` it is possible to put segment parts into holes left by earlier placements.

Use the `-Z` option when you need to keep a segment in one consecutive chunk, when you need to preserve the order of segment parts in a segment, or, more unlikely, when you need to put segments in a specific order. There can be several reasons for doing this, but most of them are fairly obscure.

The most important is to keep variables and their initializers in the same order and in one block. Compilers using UBROF 7 or later, output attributes that direct the linker to keep segment parts together, so for these compilers `-Z` is no longer required for variable initialization segments.

Use `-P` when you need to put things into several ranges, for instance when banking.

When possible, use the `-P` option instead of `-b`, since `-P` is more powerful and more convenient. The `-b` option is supported only for backward compatibility reasons.

Bit segments are always placed first, regardless of where their placement commands are given.

ADDRESS TRANSLATION

XLINK can do logical to physical address translation on output for some output formats. Logical addresses are the addresses as seen by the program, and these are the addresses used in all other XLINK command line options. Normally these addresses are also used in the output object files, but by using the `-M` option a mapping from the logical addresses to physical addresses as used in the output object file is established.

ALLOCATION SEGMENT TYPES

The following table lists the different types of segments that can be processed by XLINK:

Segment type	Description
`STACK`	Allocated from high to low addresses by default. The aligned segment size is subtracted from the load address before allocation, and successive segments are placed below the preceding segment.
`RELATIVE`	Allocated from low to high addresses by default.
`COMMON`	All segment parts are located at the same address.

Table 2: Allocation segment types

If stack segments are mixed with relative or common segments in a segment definition, the linker will produce a warning message but will allocate the segments according to the default allocation set by the first segment in the segment list.

Common segments have a size equal to the largest declaration found for the particular segment. That is, if module `A` declares a common segment `COMSEG` with size 4, while module `B` declares this segment with size 5, the latter size will be allocated for the segment.

Be careful not to overlay common segments containing code or initializers.

Relative and stack segments have a size equal to the sum of the different (aligned) declarations.

MEMORY SEGMENT TYPES

The optional *type* parameter is used for assigning a type to all of the segments in the list. The *type* parameter affects how XLINK processes the segment overlaps. Additionally, it generates information in some of the output formats that are used by some hardware emulators and by C-SPY.

Segment type	Description
BIT	Bit memory.*
CODE	Code memory.
CONST	Constant memory.
DATA	Data memory.
FAR	Data in FAR memory. XLINK will not check access to it, and a part of a segment straddling a 64 Kbyte boundary will be moved upwards to start at the boundary.
FARC, FARCONST	Constant in FAR memory (behaves as above).
FARCODE	Code in FAR memory.
HUGE	Data in HUGE memory. No straddling problems.
HUGEC, HUGECONST	Constant in HUGE memory.
HUGECODE	Code in HUGE memory.
IDATA	Internal data memory.
IDATA0	Data memory. This segment type is only used with the OKI 65000 microcontroller.
IDATA1	Internal data memory. This segment type is only used with the OKI 65000 microcontroller.
NEAR	Data in NEAR memory. Accessed using 16-bit addressing, this segment can be located anywhere in the 32-bit address space.
NEARC, NEARCONST	Constant in NEAR memory.
NPAGE	External data memory. This segment type is only used with the Mitsubishi 740 and Western Design Center 6502 microcontrollers.
UNTYPED	Default type.
XDATA	External data memory.
ZPAGE	Data memory.

Table 3: Memory segment types

* The address of a BIT segment is specified in bits, not in bytes. BIT memory is allocated first.

OVERLAP ERRORS

By default, XLINK checks to be sure that the various segments that have been defined (by the segment placement option and absolute segments) do not overlap in memory.

If any segments overlap, it will cause error 24: Segment *segment* overlaps segment *segment*. These errors can be reduced to warnings, as described in *-z*, page 38.

RANGE ERRORS

IAR compilers and assemblers can place range checks in the generated object code, to guard against violations of code-specific constraints. An example of this would be checking that the target of a relative branch is in range. These checks are performed by XLINK and result in error 18 if they fail.

Note: Range error messages are not issued for references to segments of all types. See *-R*, page 33, for more information.

EXAMPLES

To locate SEGA at address 0, followed immediately by SEGB:

```
-Z(CODE)SEGA,SEGB=0
```

To allocate SEGA downwards from FFFH, followed by SEGB below it:

```
-Z(CODE)SEGA,SEGB#FFF
```

To allocate specific areas of memory to SEGA and SEGB:

```
-Z(CODE)SEGA,SEGB=100-1FF,400-6FF,1000
```

In this example SEGA will be placed between address 100 and 1FF, if it fits in that amount of space. If it does not, XLINK will try the range 400–6FF. If none of these ranges are large enough to hold SEGA, it will start at 1000.

SEGB will be placed, according to the same rules, after segment SEGA. If SEGA fits the 100–1FF range then XLINK will try to put SEGB there as well (following SEGA). Otherwise, SEGB will go into the 400 to 6FF range if it is not too large, or else it will start at 1000.

```
-Z(NEAR)SEGA,SEGB=19000-1FFFF
```

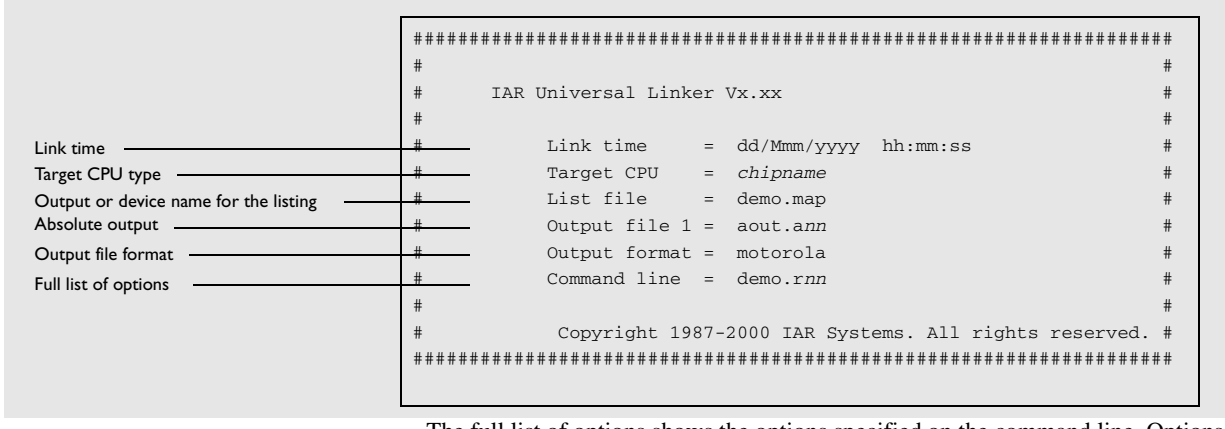
Segments SEGA and SEGB will be dumped at addresses 19000 to 1FFFF but the default 16-bit addressing mode will be used for accessing the data (i.e. 9000 to FFFF).

Listing format

The default XLINK listing consists of the sections below. Note that the examples given here are still generic. They are only used for purposes of illustration.

HEADER

Shows the command-line options selected for the XLINK command:




```
#########################################################################
#                                                                       #
#      IAR Universal Linker Vx.xx                                       #
#                                                                       #
#            Link time     =  dd/Mmm/yyyy  hh:mm:ss                     #
#            Target CPU    =  chipname                                  #
#            List file     =  demo.map                                  #
#            Output file 1 =  aout.ann                                  #
#            Output format =  motorola                                  #
#            Command line  =  demo.rnn                                  #
#                                                                       #
#             Copyright 1987-2000 IAR Systems. All rights reserved.     #
#########################################################################
```

The full list of options shows the options specified on the command line. Options in command files specified with the `-f` option are also shown, in brackets.

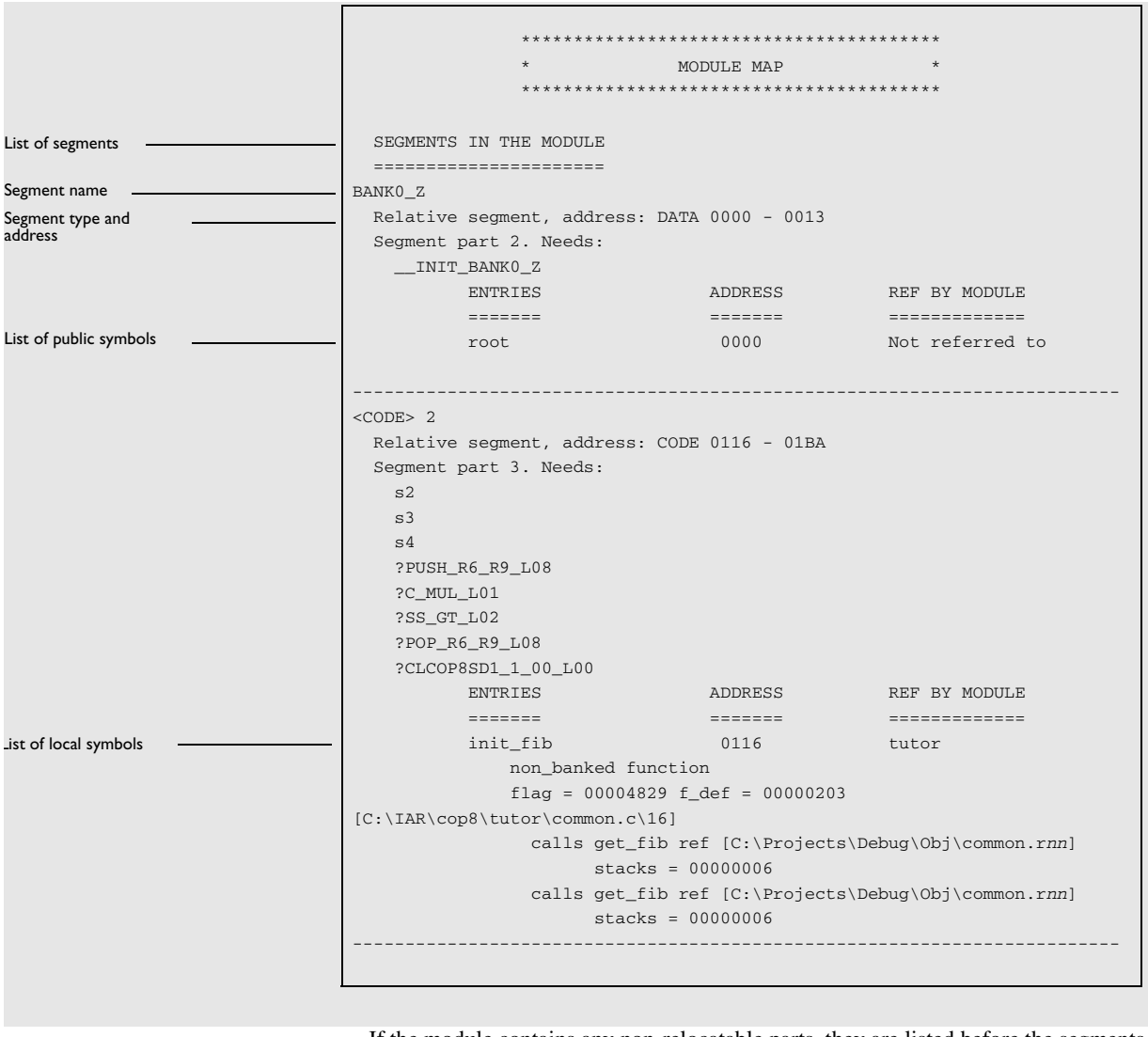
CROSS-REFERENCE

The cross-reference consists of the entry list, module map and/or the segment map. It includes the program entry point, used in some output formats for hardware emulator support; see the assembler `END` directive in the *IAR Assembler Reference Guide*.

Module map (-xm)

The module map consists of a subsection for each module that was loaded as part of the program.

Each subsection shows the following information:

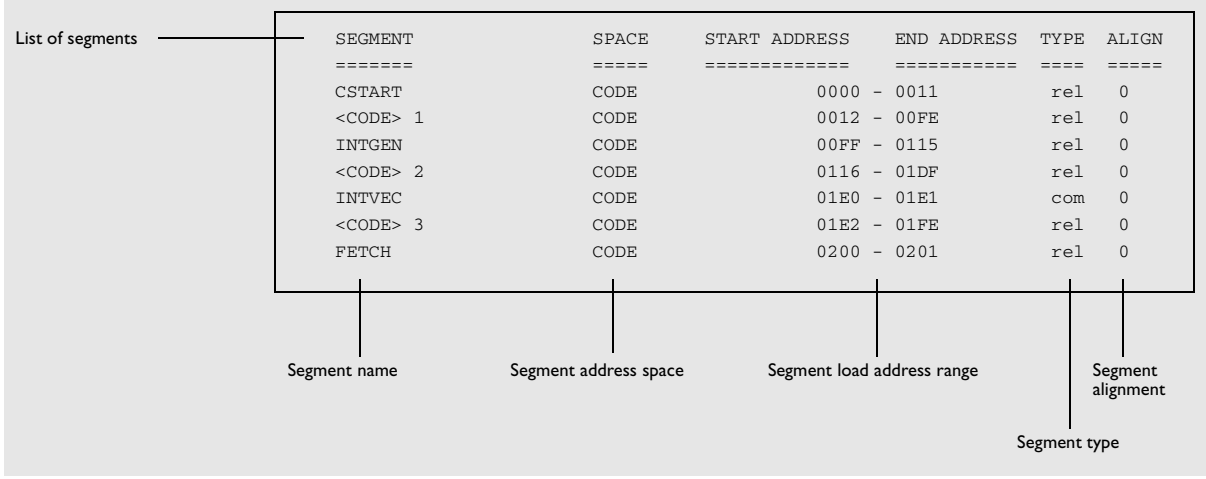



```
                ****************************************
                *              MODULE MAP              *
                ****************************************


  SEGMENTS IN THE MODULE
  ======================
BANK0_Z
  Relative segment, address: DATA 0000 - 0013
  Segment part 2. Needs:
    __INIT_BANK0_Z
           ENTRIES                ADDRESS          REF BY MODULE
           =======                =======          =============
           root                    0000            Not referred to

------------------------------------------------------------------------------
<CODE> 2
  Relative segment, address: CODE 0116 - 01BA
  Segment part 3. Needs:
    s2
    s3
    s4
    ?PUSH_R6_R9_L08
    ?C_MUL_L01
    ?SS_GT_L02
    ?POP_R6_R9_L08
    ?CLCOP8SD1_1_00_L00
           ENTRIES                ADDRESS          REF BY MODULE
           =======                =======          =============
           init_fib                0116            tutor
               non_banked function
               flag = 00004829 f_def = 00000203
[C:\IAR\cop8\tutor\common.c\16]
                 calls get_fib ref [C:\Projects\Debug\Obj\common.rnn]
                       stacks = 00000006
                 calls get_fib ref [C:\Projects\Debug\Obj\common.rnn]
                       stacks = 00000006
------------------------------------------------------------------------------
```

If the module contains any non-relocatable parts, they are listed before the segments.

Segment map (-xs)

The segment list gives the segments in increasing address order:



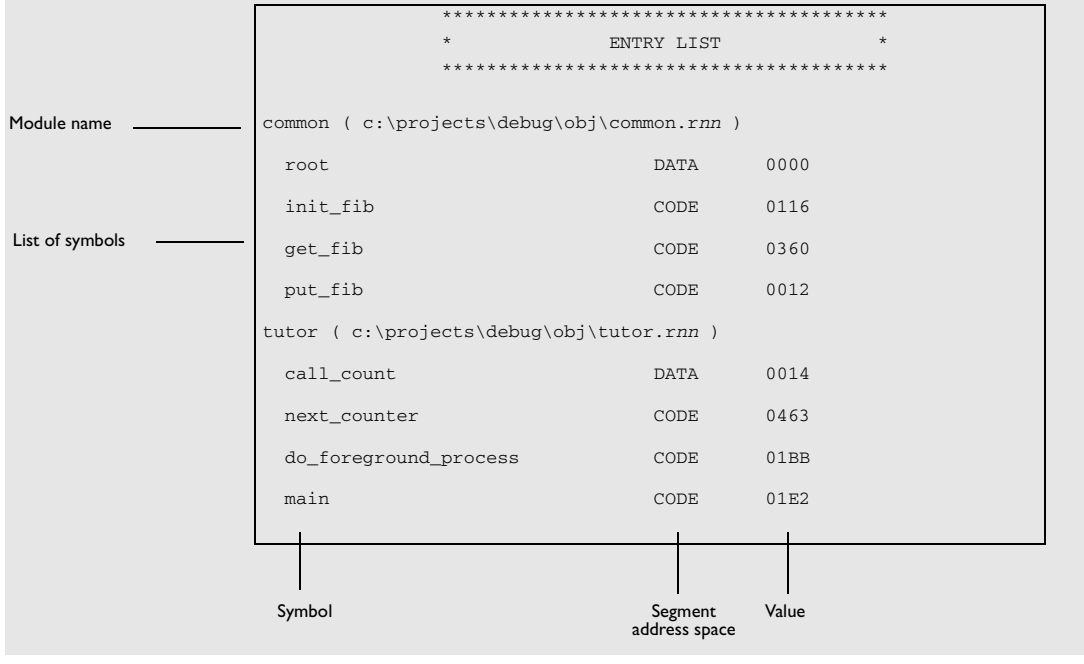
This lists the start and end address for each segment, and the following parameters:

Parameter	Description
TYPE	The type of segment: rel Relative stc Stack. bnk Banked. com Common. dse Defined but not used.
ORG	The origin; the type of segment start address: stc Absolute, for ASEG segments. flt Floating, for RSEG, COMMON, or STACK segments.
P/N	Positive/Negative; how the segment is allocated: pos Upwards, for ASEG, RSEG, or COMMON segments. neg Downwards, for STACK segment.
ALIGN	The segment is aligned to the next 2^ALIGN address boundary.

Table 4: Segment map (-xs) XLINK option

Symbol listing (-xe)

The symbol listing shows the entry name and address for each module and filename.



CHECKSUMMED AREAS AND MEMORY USAGE

If the **Generate checksum** (`-J`) and **Fill unused code memory** (`-H`) options have been specified, the listing includes a list of the checksummed areas, in order:

```
          ********************************
             CHECKSUMMED AREAS, IN ORDER
          ********************************

   00000000   -   00007FFFin CODE memory
   0000D414   -   0000D41Fin CODE memory
Checksum = 32e19
          ********************************
                END OF CROSS REFERENCE
          ********************************
   2068 bytes of CODE memory   (30700 range fill)
   2064 bytes of DATA memory   (12 range fill)
Errors: none
Warnings: none
```

This information is followed, irrespective of the options selected, by the memory usage and the number of errors and warnings.

XLINK options

The XLINK options allow you to control the operation of the IAR XLINK Linker™.

The *IAR Embedded Workbench™ User Guide* describes how to set XLINK options in the IAR Embedded Workbench, and gives reference information about the available options.

Setting XLINK options

To set options from the command line, either:

- Specify the options on the command line, after the `xlink` command.
- Specify the options in the `XLINK_ENVPAR` environment variable; see the chapter *XLINK environment variables*.
- Specify the options in an extended linker command (`xcl`) file, and include this on the command line with the `-f` *`file`* command.

Note: You can include C-style `/*...*/` or `//` comments in linker command files.

Summary of options

The following table summarizes the XLINK command line options:

Command line option	Description
`-!`	Comment delimiter
`-A` *`file`*`,…`	Loads as program
`-a`	Disables static overlay
`-B`	Always generates output
`-b`*`bank_def`*	Defines banked segments
`-C` *`file`*`, …`	Loads as library
`-c`*`cpu`*	Specifies processor type
`-D`*`symbol=value`*	Defines symbol
`-d`	Disables code generation
`-E` *`file`*`,…`	Inherent, no object code
`-e`*`new=old[,old]`* …	Renames external symbols

Table 5: XLINK options summary

Command line option	Description
`-F`*`format`*	Specifies output format
`-f` *`file`*	Specifies XCL filename
`-G`	Disables global type checking
`-H`*`hexstring`*	Fills unused code memory
`-h`*`[(seg_type)]{range}`*	Fills ranges.
`-I`*`pathname`*	Includes paths
`-J`*`size,method[,flags]`*	Generates checksum
`-K`*`segs=inc,count`*	Duplicates code
`-L`*`directory`*	Lists to directory
`-l` *`file`*	Lists to named file
`-M`*`range_def`*	Maps logical addresses to physical addresses
`-n[c]`	Ignores local symbols
`-Oformat[,variant][=filename]`	Multiple output files
`-o` *`file`*	Output file
`-P`*`pack_def`*	Defines packed segments
`-p`*`lines`*	Specifies lines/page
`-Q`	Scatter loading
`-R[w]`	Disables range check
`-r`	Debug information
`-rt`	Debug information with terminal I/O
`-S`	Silent operation
`-w[n|s|t|ID[=`*`severity`*`]]`	Sets diagnostics control
`-x[e][m][s]`	Specifies cross-reference
`-Y[`*`char`*`]`	Format variant
`-y[`*`chars`*`]`	Format variant
`-Z[@]`*`seg_def`*	Defines segments
`-z`	Segment overlap warnings

Table 5: XLINK options summary

Descriptions of XLINK options

The following sections describe each of the XLINK command line options in detail.

-!

`-! comment -!`

A `-!` can be used for bracketing off comments in an extended linker command file. Unless the `-!` is at the beginning of a line, it must be preceded by a space or tab.

Note: You can include C-style and C++style comments in your files; the use of these is recommended since they are less error-prone than `-!`.

-A

`-A file,…`

Use `-A` to temporarily force all of the modules within the specified input files to be loaded as if they were all program modules, even if some of the modules have the `LIBRARY` attribute.

This option is particularly suited for testing library modules before they are installed in a library file, since the `-A` option will override an existing library module with the same entries. In other words, XLINK will load the module from the *input file* specified in the `-A` argument instead of one with an entry with the same name in a library module.

This option is identical to the **Load as PROGRAM** option in the **XLINK** category in the IAR Embedded Workbench.

-a

`-a{i|w}[function-list]`

Use `-a` to control the static memory allocation of variables. The options are as follows:

Option	Description
`-a`	Disables overlaying totally, for debugging purposes.
`-ai`	Disables indirect tree overlaying.
`-aw`	Disables warning 16, Function is called from two function trees. Do this only if you are sure the code is correct.

Table 6: Disabling static overlay options

In addition, the -a option can specify one or more function lists, to specify additional options for specified functions. Each function list can have the following form, where *function* specifies a public function or a *module:function* combination:

Function list	Description
(*function*,*function*...)	Function trees will not be overlaid with another function.
[*function*,*function*...]	Function trees will not be allocated unless they are called by another function.
{*function*,*function*...}	Indicates that the specified functions are interrupt functions.

Table 7: Disabling static overlay function lists

Several -a options may be specified, and each -a option may include several suboptions, in any order.

`-B -B`

Use -B to generate an output file even if a non-fatal error was encountered during the linking process, such as a missing global entry or a duplicate declaration. Normally, XLINK will not generate an output file if an error is encountered.

Note: XLINK always aborts on fatal errors, even with -B specified.

The -B option allows missing entries to be patched in later in the absolute output image.

This option is identical to the **Always generate output** option in the **XLINK** category in the IAR Embedded Workbench.

`-b -b [addrtype] [(type)] segments=first,length,increment[, count]`

where the parameters are as follows:

addrtype The type of load addresses used when dumping the code:

omitted	Logical addresses with bank number.
#	Linear physical addresses.
@	64180-type physical addresses.

type Specifies the memory type for all segments if applicable for the target microcontroller. If omitted it defaults to UNTYPED.

segments The list of banked segments to be linked.
The delimiter between segments in the list determines how they are packed:

:	(colon)	The next segment will be placed in a new bank.
,	(comma)	The next segment will be placed in the same bank as the previous one.

first The start address of the first segment in the banked segment list. This is a 32-bit value: the high-order 16 bits represent the starting bank number while the low-order 16 bits represent the start address for the banks in the logical address area.

length The length of each bank, in bytes. This is a 16-bit value.

increment The incremental factor between banks, i.e. the number that will be added to *first* to get to the next bank. This is a 32-bit value: the high-order 16 bits are the bank increment, and the low-order 16 bits are the increment from the start address in the logical address area.

count Number of banks available, in decimal.

The option `-b` can be used to allocate banked segments for a program that is designed for bank-switched operation. It also enables the banking mode of linker operation. However, we recommend that you instead use `-P` to define packed segments. See page 32.

There can be more than one `-b` definition.

Logical addresses are the addresses as seen by the program. In most bank-switching schemes this means that a logical address contains a bank number in the most significant 16 bits and an offset in the least significant 16 bits.

Linear physical addresses are calculated by taking the bank number (the most significant 16 bits of the address) times the bank length and adding the offset (the least significant 16 bits of the address). Specifying linear physical addresses affects the load addresses of bytes output by XLINK, not the addresses seen by the program.

64180-type physical addresses are calculated by taking the least significant 8 bits of the bank number, shifting it left 12 bits and then adding the offset.

Using either of these simple translations is only useful for some rather simple memory layouts. Linear physical addressing as calculated by XLINK is useful for a bank memory at the very end of the address space. Anything more complicated will need some post-processing of XLINK output, either by a PROM programmer or a special program. See the `simple` subdirectory for source code for the start of such a program.

For example, to specify that the three code segments BSEG1, BSEG2, and BSEG3 should be linked into banks starting at 8000, each with a length of 4000, with an increment between banks of 10000:

```
-b(CODE)BSEG1,BSEG2,BSEG3=8000,4000,10000
```

For more information see, *Segment control*, page 8.

Note: This option is included for backward compatibility reasons. We recommend that you instead use `-P` to define packed segments. See page 32.

`-C` `-C file,…`

Use `-C` to temporarily cause all of the modules within the specified input files to be treated as if they were all library modules, even if some of the modules have the `PROGRAM` attribute. This means that the modules in the input files will be loaded only if they contain an entry that is referenced by another loaded module.

This option is identical to the **Load as LIBRARY** option in the **XLINK** category in the IAR Embedded Workbench.

`-c` `-c`*`processor`*

Use `-c` to specify the target processor.

The environment variable `XLINK_CPU` can be set to install a default for the `-c` option so that it does not have to be specified on the command line; see the chapter *XLINK environment variables*.

This option is related to the **Target** options in the **General** category in the IAR Embedded Workbench.

`-D` `-D`*`symbol=value`*

The parameter *`symbol`* is any external (`EXTERN`) symbol in the program that is not defined elsewhere, and *`value`* the value to be assigned to *`symbol`*.

Use `-D` to define absolute symbols at link time. This is especially useful for configuration purposes. Any number of symbols can be defined in a linker command file. The symbol(s) defined in this manner will belong to a special module generated by the linker called `?ABS_ENTRY_MOD`.

XLINK will display an error message if you attempt to redefine an existing symbol.

This option is identical to the **#define** option in the **XLINK** category in the IAR Embedded Workbench.

`-d` `-d`

Use `-d` to disable the generation of output code from XLINK. This option is useful for the trial linking of programs; for example, checking for syntax errors, missing symbol definitions, etc. XLINK will run slightly faster for large programs when this option is used.

`-E` `-E` *file,…*

Use `-E` to empty load specified input files; they will be processed normally in all regards by the linker but output code will not be generated for these files.

One potential use for this feature is in creating separate output files for programming multiple EPROMs. This is done by empty loading all input files except the ones you want to appear in the output file.

In the following example a project consists of four files, `file1` to `file4`, but we only want object code generated for `file4` to be put into an EPROM:

```
-E file1,file2,file3
file4
-o project.hex
```

To read object files from `v:\general\lib` and `c:\project\lib`:

```
-Iv:\general\lib;c:\project\lib\
```

This option is related to the **Input** options in the **XLINK** category in the IAR Embedded Workbench.

`-e` `-e`*new=old* `[,`*old*`]` …

Use `-e` to configure a program at link time by redirecting a function call from one function to another.

This can also be used for creating stub functions; i.e. when a system is not yet complete, undefined function calls can be directed to a dummy routine until the real function has been written.

`-F` `-F`*format*

Use `-F` to specify the output format.

The environment variable `XLINK_FORMAT` can be set to install an alternate default format on your system; see the chapter *XLINK environment variables*.

The parameter should be one of the supported XLINK output formats; for details of the formats see the chapter *XLINK output formats*.

Note: Specifying the `-F` option as `DEBUG` does not include C-SPY debug support. Use the `-r` option instead.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

`-f` `-f file`

Use `-f` to extend the XLINK command line by reading arguments from a command file, just as if they were typed in on the command line. If not specified, an extension of `xcl` is assumed.

Arguments are entered into the linker command file with a text editor using the same syntax as on the command line. However, in addition to spaces and tabs, the Enter key provides a valid delimiter between arguments. A command line may be extended by entering a backslash, `\`, at the end of line.

Note: You can include C-style `/*...*/` or `//` comments in linker command files.

This option is related to the **Include** options in the **XLINK** category in the IAR Embedded Workbench.

`-G` `-G`

Use `-G` to disable type checking at link time. While a well-written program should not need this option, there may be occasions where it is helpful.

By default, XLINK performs link-time type checking between modules by comparing the external references to an entry with the `PUBLIC` entry (if the information exists in the object modules involved). A warning is generated if there are mismatches.

This option is identical to the **No global type checking** option in the **XLINK** category in the IAR Embedded Workbench.

`-H` `-H`*`hexstring`*

Use `-H` to fill all gaps between segment parts introduced by the linker with the repeated *`hexstring`*.

The linker can introduce gaps because of alignment restrictions, or to fill ranges given in segment placement options. The normal behavior, when no `-H` option is given, is that these gaps are not given a value in the output file.

The following example will fill all the gaps with the value `0xbeef`:

```
-HBEEF
```

Even bytes will get the value `0xbe`, and odd bytes will get the value `0xef`.

This option corresponds to the **Fill unused code memory** option in the **XLINK** category in the IAR Embedded Workbench.

`-h` `-h[(seg_type)]{range}`

Use `-h` to specify the ranges to fill. Normally, all ranges given in segment-placement commands (`-Z` and `-P`) into which any actual content (code or constant data) is placed, are filled. For example:

```
-Z(CODE)INTVEC=0-FF
-Z(CODE)RCODE,CODE,SHORTAD_ID=0-7FFF,F800-FFFF
-Z(DATA)SHORTAD_I,SHORTAD_Z=8000-8FFF
```

If `INTVEC` contains anything, the range `0-FF` will be filled. If `RCODE`, `CODE` or `SHORTAD_ID` contains anything, the ranges `0-7FFF` and `F800-FFFF` will be filled. `SHORTAD_I` and `SHORTAD_Z` are normally only place holders for variables, which means that the range `8000-8FFF` will not be filled.

Using `-h` you can explicitly specify which ranges to fill. The syntax allows you to use an optional segment type (which can be used for specifying address space for architectures with multiple address spaces) and one or more address ranges. Example:

```
-h(CODE)0-FFFF
```

or, equivalently, as segment type `CODE` is the default,

```
-h0-FFFF
```

will cause the range `0-FFFF` to be filled, regardless of what ranges are specified in segment-placement commands. Often `-h` will not be needed.

The `-h` option can be specified more than once, in order to specify fill ranges for more than one address space. It does not restrict the ranges used for calculating checksums.

`-I` `-I*pathname*`

Specifies a pathname to be searched for object files.

By default, XLINK searches for object files only in the current working directory. The `-I` option allows you to specify the names of the directories which it will also search if it fails to find the file in the current working directory.

This is equivalent to the `XLINK_DFLTDIR` environment variable; see the chapter *XLINK environment variables*.

This option is related to the **Include** option in the **XLINK** category in the IAR Embedded Workbench.

-J -J*size*,*method*[,*flags*]

Use -J to checksum all generated raw data bytes. This option can only be used if the -H option has been specified.

size specifies the number of bytes in the checksum, and can be 1, 2, or 4.

method specifies the algorithm used, and can be one of the following:

Method	Description
sum	Simple arithmetic sum.
crc16	CRC16 (generating polynomial 0x11021).
crc32	CRC32 (generating polynomial 0x104C11DB7).
crc=*n*	CRC with a generating polynomial of *n*.

Table 8: Checksumming algorithms

flags can be used to specify complement and/or the bit-order of the checksum.

Flag	Description
1	Specifies one's complement.
2	Specifies two's complement.
m	Mirrored bytes. Reverses the order of the bits within each byte when calculating the checksum. You can specify just m, or m in combination with either 1 or 2.

Table 9: Checksumming flags

In all cases it is the least significant 1, 2, or 4 bytes of the result that will be output, in the natural byte order for the processor. The CRC checksum is calculated as if the following code was called for each bit in the input, with the most significant bit of each byte first as default, starting with a CRC of 0:

```
unsigned long
crc(int bit, unsigned long oldcrc)
{
   unsigned long newcrc = (oldcrc << 1) ^ bit;
   if (oldcrc & 0x80000000)
      newcrc ^= POLY;
   return newcrc;
}
```

POLY is the generating polynomial. The checksum is the result of the final call to this routine. If the m flag is specified, the checksum is calculated with each byte bit-reversed—i. e. with the least significant bit first. Notice that the whole result will also be bit-reversed. If the 1 or 2 flag is specified, the checksum is the one's or two's complement of the result.

The linker will place the checksum byte(s) at the label `__checksum` in the segment `CHECKSUM`. This segment must be placed using the segment placement options like any other segment.

For example, to calculate a 4-byte checksum using the generating polynomial `0x104C11DB7` and output the one's complement of the calculated value, specify:

```
-J4,crc32,1
```

This option corresponds to the **Generate checksum** option in the **XLINK** category in the IAR Embedded Workbench.

`-K` `-K`*`segs`*`=`*`inc`*`,`*`count`*

Use `-K` to duplicate any raw data bytes from the segments in *`segs`* *`count`* times, adding *`inc`* to the addresses each time. This will typically be used for segments mentioned in a `-Z` option.

This can be used for making part of a PROM be non-banked even though the entire PROM is physically banked. Use the `-b` or `-P` option to place the banked segments into the rest of the PROM.

For example, to copy the contents of the `RCODE0` and `RCODE1` segments four times, using addresses `0x20000` higher each time, specify:

```
-KRCODE0,RCODE1=20000,4
```

This will place 5 instances of the bytes from the segments into the output file, at the addresses `x`, `x+0x20000`, `x+0x40000`, `x+0x60000`, and `x+0x80000`.

For more information, see *Segment control*, page 8.

`-L` `-L[`*`directory`*`]`

Causes the linker to generate a listing and send it to the file *`directory`*`\`*`outputname`*`.lst`. Notice that you must not include a space before the prefix.

By default, the linker does not generate a listing. To simply generate a listing, you use the `-L` option without specifying a directory. The listing is sent to the file with the same name as the output file, but extension `lst`.

`-L` may not be used as the same time as `-l`.

This option is related to the **List** options in the **XLINK** category in the IAR Embedded Workbench.

`-l` `-l file`

Causes the linker to generate a listing and send it to the named file. If no extension is specified, `lst` is used by default. However, an extension of `map` is recommended to avoid confusing linker list files with assembler or compiler list files.

`-l` may not be used as the same time as `-L`.

This option is related to the **List** options in the **XLINK** category in the IAR Embedded Workbench.

`-M` `-M[(type)]logical_range=physical_range`

where the parameters are as follows:

type		Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.
range	*start-end*	The range starting at *start* and ending at *end*.
	[*start-end*]**count+offset*	Specifies *count* ranges, where the first is from *start* to *end*, the next is from *start+offset* to *end+offset*, and so on. The *+offset* part is optional, and defaults to the length of the range.
	[*start-end*]*/pagesize*	Specifies the entire *range* from *start* to *end*, divided into pages of size and alignment *pagesize*. *Note*: The *start* and *end* of the range do not have to coincide with a page boundary.

XLINK can do logical to physical address translation on output for some output formats. Logical addresses are the addresses as seen by the program, and these are the addresses used in all other XLINK command line options. Normally these addresses are also used in the output object files, but by using the `-M` option, a mapping from the logical addresses to physical addresses, as used in the output object file, is established.

Each occurrence of `-M` defines a linear mapping from a list of logical address ranges to a list of physical address ranges, in the order given, byte by byte. For example the command:

```
-M0-FF,200-3FF=1000-11FF,1400-14FF
```

will define the following mapping:

Logical address	Physical address
0x00-0xFF	0x1000-0x10FF
0x200-0x2FF	0x1100-0x11FF
0x300-0x3FF	0x1400-0x14FF

Table 10: Mapping logical to physical addresses (example)

Several -M command line options can be given to establish a more complex mapping.

Address translation can be useful in banked systems. The following example assumes a code bank at address 0x8000 of size 0x4000, replicated 4 times, occupying a single physical ROM. To define all the banks using physically contiguous addresses in the output file, the following command is used:

```
-P(CODE)BANKED=[8000-BFFF]*4+10000 // Place banked code
-M(CODE)[8000-BFFF]*4+10000=10000  // Single ROM at 0x10000
```

The -M option only supports some output formats, primarily the simple formats with no debug information. The following list shows the currently supported formats:

aomf80196	ashling-z80	pentica-a
aomf8051	extended-tekhex	pentica-b
aomf8096	hp-code	pentica-c
ashling	intel-extended	pentica-d
ashling-6301	intel-standard	rca
ashling-64180	millenium	symbolic
ashling-6801	motorola	ti7000
ashling-8080	mpds-code	typed
ashling-8085	mpds-symb	zax

-n -n[c]

Use -n to ignore all local (non-public) symbols in the input modules. This option speeds up the linking process and can also reduce the amount of host memory needed to complete a link. If -n is used, locals will not appear in the list file cross-reference and will not be passed on to the output file.

Use -nc to ignore just compiler-generated local symbols, such as jump or constant labels. These are usually only of interest when debugging at assembler level.

Note: Local symbols are only included in files if they were compiled or assembled with the appropriate option to specify this.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

`-O` `-O`*`format`*`[,`*`variant`*`][=`*`filename`*`]`

Use the `-O` option to create one or more output files of the *`format`* format, possibly with the *`variant`* variant (just as if you had used the `-Y` or `-y` option). If no filename is specified, the output file will be given the same name as a previously specified output file, or the name given in a `-o` option, with the default extension for the format. (Typically you would want all output files specified using the `-O` option to have the same filename.) If the first character of *`filename`* is a `.` (a period), *`filename`* is assumed to be an extension, and the file receives the same name as if no name was specified, but with the specified extension. Any number of `-O` command line options can be specified.

Example

```
-Odebug=foo
-Omotorola=.s99
-Ointel-extended,1=abs.x
```

This will result in one output file named `foo.dbg`, using the UBROF format, one named `foo.s99`, using the MOTOROLA format, and one named `abs.x`, using the INTEL-EXTENDED format just as if `-Y1` had also been specified.

Output files produced by using `-O` will be in addition to those produced by using the `-F`, `-o`, or `-y` options. This means that extra output files can be added to the linker command file despite that this feature is not supported in the IAR Embedded Workbench.

Note: If `-r` is specified—or, in the IAR Embedded Workbench, one of the XLINK options **Debug info** and **Debug info with terminal I/O**—only one output file is generated, using the UBROF format and selecting special runtime library modules for IAR C-SPY.

`-o` `-o` *`file`*

Use `-o` to specify the name of the XLINK output file. If a name is not specified, the linker will use the name `aout.hex`. If a name is supplied without a file type, the default file type for the selected output format will be used. See *-F*, page 23, for additional information.

If a format is selected that generates two output files, the user-specified file type will only affect the primary output file (first format).

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

`-P` `-P [(type)]segments=range[,range] …`

where the parameters are as follows:

type		Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.
segments		A list of one or more segments to be linked, separated by commas.
range	*start-end*	The range starting at *start* and ending at *end*.
	[*start-end*]*count+*offset*	Specifies *count* ranges, where the first is from *start* to *end*, the next is from *start+offset* to *end+offset*, and so on. The *+offset* part is optional, and defaults to the length of the range.
	[*start-end*]/*pagesize*	Specifies the entire range from *start* to *end*, divided into pages of size and alignment *pagesize*. *Note*: The *start* and *end* of the range do not have to coincide with a page boundary.

Use `-P` to pack the segment parts from the specified segments into the specified ranges, where a segment part is defined as that part of a segment that originates from one module. The linker splits each segment into its segment parts and forms new segments for each of the ranges. All the ranges must be closed; i.e. both *start* and *end* must be specified. The segment parts will not be placed in any specific order into the ranges.

The following examples show the address range syntax:

`0-9F,100-1FF`	Two ranges, one from zero to `9F`, one from `100` to `1FF`.
`[1000-1FFF]*3+2000`	Three ranges: `1000-1FFF,3000-3FFF,5000-5FFF`.
`[1000-1FFF]*3`	Three ranges: `1000-1FFF,2000-2FFF,3000-3FFF`.

`[50-77F]/200` **Five ranges:** `50-1FF,200-3FF,400-5FF,600-77F`.

All numbers in segment placement command line options are interpreted as hexadecimal unless they are preceded by a `.` (period). That is, the numbers written as `10` and `.16` are both interpreted as sixteen.

For more information see *Segment control*, page 8.

`-p -p`*`lines`*

Sets the number of lines per page for the XLINK list files to *`lines`*, which must be in the range 10 to 150.

The environment variable `XLINK_PAGE` can be set to install a default page length on your system; see the chapter *XLINK environment variables*.

This option is related to the **List** options in the **XLINK** category in the IAR Embedded Workbench.

`-Q -Q`*`segment=initializer_segment`*

Use `-Q` to do automatic setup for copy initialization of segments (scatter loading). This will cause the linker to generate a new segment (*`initializer_segment`*) into which it will place all data content of the segment *`segment`*. Everything else, e.g. symbols and debugging information, will still be associated with the segment *`segment`*. Code in the application must at runtime copy the contents of *`initializer_segment`* (in ROM) to *`segment`* (in RAM).

This is very similar to what compilers do for initialized variables and is useful for code that needs to be in RAM memory.

The segment *`initializer_segment`* must be placed like any other segment using the segment placement commands.

Assume for example that the code in the segment `RAMCODE` should be executed in RAM. `-Q` can be used for making the linker transfer the contents of the segment `RAMCODE` (which will reside in RAM) into the (new) segment `ROMCODE` (which will reside in ROM), like this:

```
-QRAMCODE=ROMCODE
```

Then `RAMCODE` and `ROMCODE` need to be placed, using the usual segment placement commands. `RAMCODE` needs to be placed in the relevant part of RAM, and `ROMCODE` in ROM. Here is an example:

```
-Z(DATA)RAM segments,RAMCODE,Other RAM=0-1FFF
-Z(CODE)ROM segments,ROMCODE,Other ROM segments=4000-7FFF
```

This will reserve room for the code in `RAMCODE` somewhere between address `0` and address `0x1FFF`, the exact address depending on the size of other segments placed before it. Similarly, `ROMCODE` (which now contains all the original contents of `RAMCODE`) will be placed somewhere between `0x4000` and `0x7FFF`, depending on what else is being placed into ROM.

At some time before executing the first code in `RAMCODE`, the contents of `ROMCODE` will need to be copied into it. This can be done as part of the startup code (in `CSTARTUP`) or in some other part of the code.

`-R` `-R[w]`

Use `-R` to specify the address range check.

If an address is relocated out of the target CPU's address range (code, external data, or internal data address) an error message is generated. This usually indicates an error in an assembly language module or in the segment placement.

The following table shows how the modifiers are mapped:

Option	Description
(default)	An error message is generated.
`-Rw`	Range errors are treated as warnings
`-R`	Disables the address range checking

Table 11: Disable range check options

Note: Range error messages are never issued for references to segments of any of the following types:

```
NEAR
NEARC, NEARCONST
NEARCODE
FAR
FARC, FARCONST
FARCODE
HUGE
HUGEC, HUGECONST
HUGECODE
```

All banked segments.

This option is related to the **Range checks** options in the **XLINK** category in the IAR Embedded Workbench.

`-r` `-r`

Use `-r` to output a file in DEBUG (UBROF) format, with a `dnn` extension, to be used with the IAR C-SPY Debugger. For emulators that support the IAR Systems DEBUG format, use `-F ubrof`.

Specifying `-r` overrides any `-F` option.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

`-rt` `-rt`

Use `-rt` to use the output file with the IAR C-SPY Debugger and emulate terminal I/O.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

`-S` `-S`

Use `-S` to turn off the XLINK sign-on message and final statistics report so that nothing appears on your screen during execution. However, this option does not disable error and warning messages or the list file output.

`-w` `-w[n|s|t|ID[=severity]]`

Use just `-w` without an argument to suppress warning messages.

The optional argument `n` specifies which warning to disable; for example, to disable warnings 3 and 7:

```
-w3 -w7
```

Specifying `-ws` changes the return status of XLINK as follows:

Condition	Default	-ws
No errors or warnings	0	0
Warnings but no errors	0	1
One or more errors	2	2

Table 12: Diagnostic control conditions (-ws)

Specifying `-wt` suppresses the detailed type information given for warnings 6 (type conflict) and 35 (multiple structs with the same tag).

Specifying `-wID` changes the severity of a particular diagnostic message. `ID` is the identity of a diagnostic message, which is either the letter `e` followed by an error number, the letter `w` followed by a warning number, or just a warning number.

The optional argument `severity` can be either `i`, `w`, or `e`. If omitted it defaults to `i`.

Severity	Description
`i`	Ignore this diagnostic message. No diagnostic output.
`w`	Report this diagnostic message as a warning.
`e`	Report this diagnostic message as an error.

Table 13: Changing diagnostic message severity

`-w` can be used several times in order to change the severity of more than one diagnostic message

Fatal errors are not affected by this option.

Some examples:

```
-w26
-ww26
-ww26=i
```

These three are equivalent and turn off warning 26.

```
-we106=w
```

This causes error 106 to be reported as a warning.

If the argument is omitted, all warnings are disabled.

As the severity of diagnostic messages can be changed, the identity of a particular diagnostic message includes its original severity as well as its number. That is, diagnostic messages will typically be output as:

`Warning[w6]`: Type conflict for external/entry ...

`Error[e1]`: Undefined external ...

This option is related to the **Diagnostics** options in the **XLINK** category in the IAR Embedded Workbench.

-x `-x[e][m][s]`

Use `-x` to include a segment map in the XLINK list file. This option is used with the list options `-L` or `-l`. See page 27 for additional information.

The following modifiers are available:

Modifier	Description
`s`	A list of all the segments in dump order.
`e`	An abbreviated list of every entry (global symbol) in every module. This entry map is useful for quickly finding the address of a routine or data element.
`m`	A list of all segments, local symbols, and entries (public symbols) for every module in the program.

Table 14: Cross-reference options

When the `-x` option is specified without any of the optional parameters, a default cross-reference list file will be generated which is equivalent to `-xms`. This includes:

- A header section with basic program information.
- A module load map with symbol cross-reference information.
- A segment load map in dump order.

Cross-reference information is listed to the screen if neither of the `-l` or `-L` options has been specified.

This option is related to the **List** options in the **XLINK** category in the IAR Embedded Workbench.

-Y `-Y[char]`

Use `-Y` to select enhancements available for some output formats. For more information, see the chapter *XLINK output formats*.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-y `-y[chars]`

Use`-y` to specify output format variants for some formats. A sequence of flag characters can be specified after the option `-y`. The affected formats are `ELF`, `IEEE695`, and `XCOFF78K`.

For more information, see the chapter *XLINK output formats*.

This option is related to the **Output** options in the **XLINK** category in the IAR Embedded Workbench.

-Z `-Z[@][(type)]segments[=|#]range[,range]` ...

The parameters are as follows:

`@`		Allocates the segments without taking into account any other use of the address ranges given. This is useful if you for some reason want the segments to overlap.
type		Specifies the memory type for all segments if applicable for the target processor. If omitted it defaults to `UNTYPED`.
segments		A list of one or more segments to be linked, separated by commas. The segments are allocated in memory in the same order as they are listed. Appending `+`*nnnn* to a segment name increases the amount of memory that XLINK will allocate for that segment by *nnnn* bytes.
= or #	Specifies how segments are allocated:	
	=	Allocates the segments so they begin at the start of the specified range (upward allocation).
	#	Allocates the segment so they finish at the end of the specified range (downward allocation).
	If an allocation operator (and range) is not specified, the segments will be allocated upwards from the last segment that was linked, or from address 0 if no segments have been linked.	
range	*start-end*	The range starting at *start* and ending at *end*.
	[*start-end*]*count+offset*	Specifies *count* ranges, where the first is from *start* to *end*, the next is from *start+offset* to *end+offset*, and so on. The *+offset* part is optional, and defaults to the length of the range.

`[`*`start-end`*`]/`*`pagesize`	Specifies the entire *`range`* from *`start`* to *`end`*, divided into pages of size and alignment *`pagesize`*. *Note*: The *`start`* and *`end`* of the range do not have to coincide with a page boundary.

Use `-Z` to specify how and where segments will be allocated in the memory map.

If the linker finds a segment in an input file that is not defined either with `-Z`, `-b`, or `-P`, an error is reported. There can be more than one `-Z` definition.

Placement into far memory (the `FAR`, `FARCODE`, `FARCONST` segment types) is treated separately. Using the `-Z` option for far memory, places the segments that fit entirely into the first page and range sequentially, and then places the rest using a special variant of sequential placement that can move an individual segment part into the next range if it did not fit. This means that far segments can be split into several memory ranges, but it is guaranteed that a far segment has a well-defined start and end.

The following examples show the address range syntax:

`0-9F,100-1FF`	Two ranges, one from zero to `9F`, one from `100` to `1FF`.
`[1000-1FFF]*3+2000`	Three ranges: `1000-1FFF`,`3000-3FFF`,`5000-5FFF`.
`[1000-1FFF]*3`	Three ranges: `1000-1FFF`,`2000-2FFF`,`3000-3FFF`.
`[50-77F]/200`	Five ranges: `50-1FF`,`200-3FF`,`400-5FF`,`600-77F`.

All numbers in segment placement command-line options are interpreted as hexadecimal unless they are preceded by a `.` (period). That is, the numbers written as `10` and `.16` are both interpreted as sixteen.

For more information see *Segment control*, page 8.

`-z` `-z`

Use `-z` to reduce segment overlap errors to warnings, making it possible to produce cross-reference maps, etc.

This option is related to the **Diagnostics** options in the **XLINK** category in the IAR Embedded Workbench.

XLINK output formats

This chapter gives a summary of the IAR XLINK Linker™ output formats.

Single output file

The following formats result in the generation of a single output file:

Format	Type	Extension	Address type
AOMF8051	binary	from CPU	N
AOMF8096	binary	from CPU	N
AOMF80196	binary	from CPU	N
AOMF80251	binary	from CPU	N
ASHLING	binary	none	N
ASHLING-6301	binary	from CPU	N
ASHLING-64180	binary	from CPU	NS
ASHLING-6801	binary	from CPU	N
ASHLING-8080	binary	from CPU	NS
ASHLING-8085	binary	from CPU	NS
ASHLING-Z80	binary	from CPU	NS
DEBUG (UBROF)	binary	dbg	NL
ELF*	binary	elf	NL
EXTENDED-TEKHEX	ASCII	from CPU	NLPS
HP-CODE	binary	x	NLPS
HP-SYMB	binary	l	NLPS
IEEE695*	binary	695	NL
INTEL-EXTENDED	ASCII	from CPU	NLPS
INTEL-STANDARD	ASCII	from CPU	N
MILLENIUM (Tektronix)	ASCII	from CPU	N
MOTOROLA**	ASCII	from CPU	NLPS
MOTOROLA-S19**	ASCII	from CPU	NLPS
MOTOROLA-S28**	ASCII	from CPU	NLPS
MOTOROLA-S37**	ASCII	from CPU	NLPS

Table 15: XLINK formats generating a single output file

Format	Type	Extension	Address type
MPDS-CODE	binary	tsk	N
MPDS-SYMB	binary	sym	NLPS
MSD	ASCII	sym	N
MSP430_TXT	ASCII	txt	NLPS
NEC-SYMBOLIC	ASCII	sym	N
NEC2-SYMBOLIC	ASCII	sym	N
NEC78K-SYMBOLIC	ASCII	sym	N
PENTICA-A	ASCII	sym	NLPS
PENTICA-B	ASCII	sym	NLPS
PENTICA-C	ASCII	sym	NLPS
PENTICA-D	ASCII	sym	NLPS
RCA	ASCII	from CPU	N
SIMPLE	binary	raw	NLPS
SYMBOLIC	ASCII	from CPU	NLPS
SYSROF	binary	abs	NLPS
TEKTRONIX (Millenium)	ASCII	hex	N
TI7000 (TMS7000)	ASCII	from CPU	N
TYPED	ASCII	from CPU	NLPS
UBROF†	binary	dbg	NL
UBROF5	binary	dbg	NL
UBROF6	binary	dbg	NL
UBROF7	binary	dbg	NL
UBROF8	binary	dbg	NL
UBROF9	binary	dbg	NL
XCOFF78k*	binary	lnk	NL
ZAX	ASCII	from CPU	NLPS

Table 15: XLINK formats generating a single output file

* The format is supported only for certain CPUs and debuggers. See `xlink.htm` and `xman.htm` for more information.

** The `MOTOROLA` output format uses a mixture of the record types `S1`, `S2`, `S3` (any number of each), `S7`, `S8`, and `S9` (only one of these record types can be used, and no more than once), depending on the range of addresses output.

XLINK can generate three variants of the `MOTOROLA` output format, each using only a specific set of record types:

`MOTOROLA-S19` uses the `S1` and `S9` record types, which use 16-bit addresses.

`MOTOROLA-S28` uses the `S2` and `S8` record types, which use 24-bit addresses.

`MOTOROLA-S37` uses the `S3` and `S7` record types, which use 32-bit addresses.

† Using `-FUBROF` (or `-FDEBUG`) will generate UBROF output matching the latest UBROF format version in the input. Using `-FUBROF5` – `-FUBROF9` will force output of the specified version of the format, irrespective of the input.

Address type

The address type is one of the following:

N = Non-banked address.
L = Banked logical address.
P = Banked physical address.
S = Banked 64180 physical address.

UBROF VERSIONS

XLINK reads all UBROF versions from UBROF 3 onwards, and can output all UBROF versions from UBROF 5 onwards. There is also support for outputting something called *Old UBROF* which is an early version of UBROF 5, close to UBROF 4. See *Output format variants*, page 43.

Normally, XLINK outputs the same version of UBROF that is used in its input files, or the latest version if more than one version is found. If you have a debugger that does not support this version of UBROF, XLINK can be directed to use another version. See *-F*, page 23.

For the IAR C-SPY Debugger, this is not a problem. The command line option `-r`—which in addition to specifying UBROF output also selects C-SPY-specific library modules from the IAR standard library—always uses the same UBROF version as found in the input.

Debug information loss

When XLINK outputs a version of UBROF that is earlier than the one used in its input, there is almost always some form of debug information loss, though this is often minor.

This debug information loss can consist of some of the following items:

UBROF version	Information that cannot be fully represented in earlier versions
5	Up to 16 memory keywords resulting in different pointer types and different function calling conventions.
6	Source in header files. Assembler source debug.
7	Support for up to 255 memory keywords. Support for target type and object attributes. Enum constants connected to enum types. Arrays with more than 65535 elements. Anonymous structs/unions. Slightly more expressive variable tracking info.
8	Embedded C++ object names. Added base types. Typedefs used in the actual types. Embedded C++ types: references and pointers to members. Class members. Target defined base types.
9	Call frame information. Function call step points. Inlined function instances.

Table 16: Possible information loss with UBROF version mismatch

In each case, XLINK attempts to convert the information to something that is representable in an earlier version of UBROF, but this conversion is by necessity incomplete and can cause inconsistencies. However, in most cases the result is almost indistinguishable from the original as far as debugging is concerned.

Two output files

The following formats result in the generation of two output files:

Format	Code format	Ext.	Symbolic format	Ext.
DEBUG-MOTOROLA	DEBUG	a*nn*	MOTOROLA	obj
DEBUG-INTEL-EXT	DEBUG	a*nn*	INTEL-EXT	hex
DEBUG-INTEL-STD	DEBUG	a*nn*	INTEL-STD	hex
HP	HP-CODE	x	HP-SYMB	l
MPDS	MPDS-CODE	tsk	MPDS-SYMB	sym
MPDS-I	INTEL-STANDARD	hex	MPDS-SYMB	sym

Table 17: XLINK formats generating two output files

Format	Code format	Ext.	Symbolic format	Ext.
MPDS-M	Motorola	s19	MPDS-SYMB	sym
MSD-I	INTEL-STANDARD	hex	MSD	sym
MSD-M	Motorola	hex	MSD	sym
MSD-T	MILLENIUM	hex	MSD	sym
NEC	INTEL-STANDARD	hex	NEC-SYMB	sym
NEC2	INTEL-STANDARD	hex	NEC2-SYMB	sym
NEC78K	INTEL-STANDARD	hex	NEC2-SYMB	sym
PENTICA-AI	INTEL-STANDARD	obj	Pentica-a	sym
PENTICA-AM	Motorola	obj	Pentica-a	sym
PENTICA-BI	INTEL-STANDARD	obj	Pentica-b	sym
PENTICA-BM	Motorola	obj	Pentica-b	sym
PENTICA-CI	INTEL-STANDARD	obj	Pentica-c	sym
PENTICA-CM	Motorola	obj	Pentica-c	sym
PENTICA-DI	INTEL-STANDARD	obj	Pentica-d	sym
PENTICA-DM	Motorola	obj	Pentica-d	sym
ZAX-I	INTEL-STANDARD	hex	ZAX	sym
ZAX-M	Motorola	hex	ZAX	sym

Table 17: XLINK formats generating two output files (Continued)

Output format variants

The following enhancements can be selected for the specified output formats, using the **Format variant** (`-Y`) option:

Output format	Option	Description
PENTICA-A,B,C,D and MPDS-SYMB	YO	Symbols as *module:symbolname*.
	Y1	Labels and lines as *module:symbolname*.
	Y2	Lines as *module:symbolname*.
AOMF8051	Y0	Extra type of information for Hitex.
INTEL-STANDARD	Y0	End only with :00000001FF.
	Y1	End with PGMENTRY, else :0000001FF.
MPDS-CODE	YO	Fill with 0xFF instead.
DEBUG, -r	Y#	Old UBROF version.

Table 18: XLINK output format variants

Output format	Option	Description
INTEL-EXTENDED	Y0	Segmented variant.
	Y1	32-bit linear variant.

Table 18: XLINK output format variants (Continued)

Refer to the file `xlink.htm` for information about additional options that may have become available since this guide was published.

Use **Format variant** (`-y`) to specify output format variants for some formats. A sequence of flag characters can be specified after the option `-y`. The affected formats are `IEEE695` (see page 44), `ELF` (see page 45), and `XCOFF78K` (see page 47).

IEEE695

For `IEEE695` the available format modifier flags are:

Modifier	Description
No #define constants (`-yd`)	Do not emit any #define constant records. This can sometimes drastically reduce the size of the output file.
Output global types globally (`-yg`)	Output globally visible types in a `BB2` block at the beginning of the output file.
Output global types in each module (`-yl`)	Output the globally visible types in a BB1 block at the beginning of each module in the output file.
Treat bit sections as byte sections (`-yb`)	XLINK supports the use of IEEE-695 *based variables* to represent bit variables, and the use of bit addresses for bit-addressable sections. Turning on this modifier makes XLINK treat these as if they were byte variables or sections.
Adjust output for the Mitsubishi PDB30 debugger (`-ym`)	Turning on this modifier adjusts the output in some particular ways for the Mitsubishi PDB30 debugger. *Note*: You will need to use the `l` and `b` modifiers as well (`-ylbm`).
No block-local constants (`-ye`)	Using this modifier will cause XLINK to not emit any block-local constant in the output file. One way these can occur is if an `enum` is declared in a block.
Handle variable life times (`-yv`)	Use the *variable life time* support in IEEE-695 to output more accurate debug information for variables whose location vary.
Output stack adjust records (`-ys`)	Output IEEE-695 *stack adjust* records to indicate the offset from the stack pointer of a virtual frame pointer.

Table 19: IEEE695 format modifier flags

Modifier	Description
Output module locals in `BB10` block (`-ya`)	Output information about module local symbols in `BB10` (assembler level) blocks as well as in the `BB3` (high level) blocks, if any.
Last return refers to end of function (`-yr`)	Change the source line information for the last return statement in a function to refer to the last line of the function instead of the line where it is located.

Table 19: IEEE695 format modifier flags (Continued)

The following table shows the recommended format variant modifiers for specific debuggers:

Debugger	Format variant modifier
6812 Noral debugger	`-ygvs`
68HC16 Microtek debugger	`-ylb`
740 Mitsubishi PD38	`-ylbma`
7700 HP RTC debugger	`-ygbr`
7700 Mitsubishi PD77	`-ylbm`
H8300 HP RTC debugger	`-ygbr`
H8300H HP RTC debugger	`-ygbr`
H8S HP RTC debugger	`-ygbr`
M16C HP RTC debugger	`-ygbr`
M16C Mitsubishi PD30/PDB30/KDB30	`-ylbm`
T900 Toshiba RTE900 m25	`-ygbe`
T900 Toshiba RTE900 m15	`-ygbed`

Table 20: Format variant modifiers for specific debuggers

ELF

For ELF the available format modifier flags are:

Modifier	Description
Format suitable for debuggers from ARM Ltd. (also sets `-p` flag) (`-ya`)	Adjusts the output to suit ARM Ltd. debuggers. This changes the flag values for some debug sections in ELF and pads all sections to an even multiple of four bytes. It also has the effect of setting the `-yp` option.
Use `address_class` attributes for pointer types (`-yc`)	Outputs an `address_class` attribute for pointer types based on the UBROF memory attribute number. This format variant option requires a DWARF reader (debugger) that understands these attributes.

Table 21: ELF format modifier flags

Modifier	Description
Suppress DWARF Call Frame Information (`-yf`)	Prevents the output of a `.debug_frame` section. Note that a `.debug_frame` section is only generated if enough information is present in the linker input files.
Output types in each compilation unit, instead of once for all (`-ym`)	Normally, all types are output once, in the first compilation unit, and global debug info references are used to refer to them in the rest of the debug information. If `-ym` is specified, all types are output in each compilation unit, and compilation unit relative references are used to refer to them.
Suppress DWARF debug output (`-yn`)	Output an ELF file without debug information.
Multiple ELF program sections (`-yp`)	Output one ELF program section for each segment, instead of one section for all segments combined.
`Ref_addr` (global refs) use .`debug_info` (not file) offsets (`-ys`)	Normally, global debug info references (used for references to type records when `-ym` is not specified) are offsets into the entire file, in compliance with the DWARF specification. Specifying `-ys` causes XLINK to use `.debug_info` section offsets for these references, instead. This was the default behavior in previous XLINK versions (up to version 4.51R).
Use variant `use_location` semantics for member pointers (`-yv`)	The DWARF standard specifies a `use_location` semantics that requires passing complete objects on the DWARF expression stack, which is ill-defined. Specifying this option causes XLINK to emit `use_location` attributes where the addresses of the objects are passed instead. This format variant option requires a DWARF reader (debugger) that understands these attributes.

Table 21: ELF format modifier flags (Continued)

The XLINK ELF/DWARF format output includes module-local symbols. The command line option `-n` can be used for suppressing module-local symbols in any output format.

The XLINK output conforms to ELF as described in *Executable and Linkable Format (ELF)* and to DWARF version 2, as described in *DWARF Debugging Information Format,* revision 2.0.0 (July 27, 1993); both are parts of the Tools Interface Standard Portable Formats Specification, version 1.1.

Note: The ELF format is currently supported for the 68HC11, 68HC12, 68HC16, ARM®, MC80, M32C, SH, and V850 products.

XCOFF78K

For `XCOFF78K` the available format modifier flags are:

Modifier	Description
`-ys`	Truncates names longer than 31 characters to 31 characters. Irrespective of the setting of this modifier, section names longer than 7 characters are always truncated to 7 characters and module names are truncated to 31 characters.
`-yp`	Strips source file paths, if there are any, from source file references, leaving only the file name and extension.
`-ye`	Includes module enums. Normally XLINK does not output module-local constants in the `XCOFF78K` file. The way IAR compilers currently work these include all `#define` constants as well as all `SFR`s. Use this modifier to have them included.
`-yl`	Hobbles line number info. When outputting debug information, use this modifier to ignore any source file line number references that are not in a strictly increasing order within a function.

Table 22: XCOFF78K format modifiers

If you want to specify more than one flag, all flags must be specified after the same `-y` option; for example, `-ysp`.

Restricting the output to a single address space

It is possible to restrict output in the simple ROM output formats—intel-standard, intel-extended, motorola, motorola-s19, motorola-s28, motorola-s37, millenium, ti7000, rca, tektronix, extended-tekhex, hp-code, and mpds-code—to include only bytes from a single address space. You do this by prefixing a segment type in parentheses to the format variant. This segment type specifies the desired address space. This feature is particularly useful when used in combination with the multiple output files option, see *-O*, page 30.

Example

```
-Ointel-extended,(CODE)=file1
-Ointel-extended,(DATA)=file2
```

This will result in two output files, both using the INTEL-EXTENDED output format. The first (named `file1`) will contain only bytes in the address space used for the `CODE` segment type, while the second (named `file2`) will contain only bytes in the address space used for the `DATA` segment type. If these address spaces are not the same, the content of the two files will be different.

XLINK environment variables

The IAR XLINK Linker™ supports a number of environment variables. These can be used for creating defaults for various XLINK options so that they do not have to be specified on the command line.

Except for the `XLINK_ENVPAR` environment variable, the default values can be overruled by the corresponding command line option. For example, the `-FMPDS` command line argument will supersede the default format selected with the `XLINK_FORMAT` environment variable.

Summary of XLINK environment variables

The following environment variables can be used by the IAR XLINK Linker:

Environment variable	Description
`XLINK_COLUMNS`	Sets the number of columns per line.
`XLINK_CPU`	Sets the target CPU type.
`XLINK_DFLTDIR`	Sets a path to a default directory for object files.
`XLINK_ENVPAR`	Creates a default XLINK command line.
`XLINK_FORMAT`	Sets the output format.
`XLINK_PAGE`	Sets the number of lines per page.

Table 23: XLINK environment variables

`XLINK_COLUMNS`

Sets the number of columns per line.

Use `XLINK_COLUMNS` to set the number of columns in the list file. The default is 80 columns.

Example

To set the number of columns to 132:

```
set XLINK_COLUMNS=132
```

XLINK_CPU

Sets the target processor.

Use XLINK_CPU to set a default for the -c option so that it does not have to be specified on the command line.

Example

To set the target processor to *Chipname*:

```
set XLINK_CPU=chipname
```

Related commands

This is equivalent to the XLINK -c option; see *-c*, page 22.

XLINK_DFLTDIR

Sets a path to a default directory for object files.

Use XLINK_DFLTDIR to specify a path for object files. The specified path, which should end with \, is prefixed to the object filename.

Example

To specify the path for object files as c:\iar\lib:

```
set XLINK_DFLTDIR=c:\iar\lib\
```

XLINK_ENVPAR

Creates a default XLINK command line.

Use XLINK_ENVPAR to specify XLINK commands that you want to execute each time you run XLINK.

Example

To create a default XLINK command line:

```
set XLINK_ENVPAR=-FMOTOROLA
```

Related commands

For more information about reading linker commands from a file, see *-f*, page 24.

XLINK_FORMAT

Sets the output format.

Use XLINK_FORMAT to set the format for linker output. For a list of the available output formats, see the chapter *XLINK output formats*.

Example

To set the output format to Motorola:

```
set XLINK_FORMAT=MOTOROLA
```

Related commands

This is equivalent to the XLINK `-F` option; see *-F*, page 23.

`XLINK_PAGE` Sets the number of lines per page.

Use `XLINK_PAGE` to set the number of lines per page (20–150). The default is a list file without page breaks.

Examples

To set the number of lines per page to 64:

```
set XLINK_PAGE=64
```

Related commands

This is equivalent to the XLINK `-p` option; see *-p*, page 32.

XLINK_PAGE

XLINK diagnostics

This chapter describes the errors and warnings produced by the IAR XLINK Linker™.

Introduction

The error messages produced by the IAR XLINK Linker fall into the following categories:

- XLINK error messages.
- XLINK warning messages.
- XLINK fatal error messages.
- XLINK internal error messages.

XLINK WARNING MESSAGES

XLINK warning messages will appear when the linker detects something that may be wrong. The code that is generated may still be correct.

XLINK ERROR MESSAGES

XLINK error messages are produced when the linker detects that something is incorrect. The linking process will be aborted unless the **Always generate output** (`-B`) option is specified. The code produced is almost certainly faulty.

XLINK FATAL ERROR MESSAGES

XLINK fatal error messages abort the linking process. They occur when continued linking is useless, i.e. the fault is irrecoverable.

XLINK INTERNAL ERROR MESSAGES

During linking, a number of internal consistency checks are performed. If any of these checks fail, XLINK will terminate after giving a short description of the problem. These errors will normally not occur, but if they do you should report them to the IAR Systems Technical Support group. Please include information enough to reproduce the problem from both source and object code. This would typically include:

- The exact internal error message text.
- The object code files, as well as the corresponding source code files, of the program that generated the internal error. If the file size total is very large, please contact IAR Technical Support before sending the files.

- A list of the compiler/assembler and linker options that were used when the internal error occurred, including the linker command file. If you are using the IAR Embedded Workbench, these settings are stored in the `prj`/`pew` and `dtp` files of your project. See the *IAR Embedded Workbench™ User Guide* for information about how to view and copy that information.
- Product names and version numbers of the IAR Systems development tools that were used.

Error messages

If you get a message that indicates a corrupt object file, reassemble or recompile the faulty file since an interrupted assembly or compilation may produce an invalid object file.

The following table lists the IAR XLINK Linker error messages:

0 **Format chosen cannot support banking**
Format unable to support banking.

1 **Corrupt file. Unexpected end of file in module** *module* **(***file***) encountered**
Linker aborts immediately. Recompile or reassemble, or check the compatibility between the linker and C compiler.

2 **Too many errors encountered (>100)**
Linker aborts immediately.

3 **Corrupt file. Checksum failed in module** *module* **(***file***). Linker checksum is** *linkcheck***, module checksum is** *modcheck*
Linker aborts immediately. Recompile or reassemble.

4 **Corrupt file. Zero length identifier encountered in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

5 **Address type for CPU incorrect. Error encountered in module** *module* **(***file***)**
Linker aborts immediately. Check that you are using the right files and libraries.

6 **Program module** *module* **redeclared in file** *file***. Ignoring second module**
XLINK will not produce code unless the **Always generate output** (`-B`) option (forced dump) is used.

7 **Corrupt file. Unexpected UBROF – format end of file encountered in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

8 **Corrupt file. Unknown or misplaced tag encountered in module** *module* **(***file***). Tag** *tag*
Linker aborts immediately. Recompile or reassemble.

9 **Corrupt file. Module** *module* **start unexpected in file** *file*
Linker aborts immediately. Recompile or reassemble.

10 **Corrupt file. Segment no.** *segno* **declared twice in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

11 **Corrupt file. External no.** *ext no* **declared twice in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

12 **Unable to open file** *file*
Linker aborts immediately. If you are using the command line, check the environment variable `XLINK_DFLTDIR`.

13 **Corrupt file. Error tag encountered in module** *module* **(***file***)**
A UBROF error tag was encountered. Linker aborts immediately. Recompile or reassemble.

14 **Corrupt file. Local** *local* **defined twice in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

15 **This is no error message with this number.**

16 **Segment** *segment* **is too long for segment definition**
The segment defined does not fit into the memory area reserved for it. Linker aborts immediately.

17 **Segment** *segment* **is defined twice in segment definition -Z**`segdef`
Linker aborts immediately.

18 **Range error in module** *module* **(***file***), segment** *segment* **at address** *address***. Value** *value***, in tag** *tag***, is out of bounds**
The address is out of the CPU address range. Locate the cause of the problem using the information given in the error message.

The check can be suppressed by the `-R` option.

19 **Corrupt file. Undefined segment referenced in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

20 **Undefined external referenced in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

21 **Segment** *segment* **in module** *module* **does not fit bank**
The segment is too long. Linker aborts immediately.

22 **Paragraph no. is not applicable for the wanted CPU. Tag encountered in module** *module* **(***file***)**
Linker aborts immediately. Delete the paragraph number declaration in the `xcl` file.

23 **Corrupt file. T_REL_FI_8 or T_EXT_FI_8 is corrupt in module** *module* **(***file***)**
The tag `T_REL_FI_8` or `T_EXT_FI_8` is faulty. Linker aborts immediately. Recompile or reassemble.

24 **Segment** *segment* **overlaps segment** *segment*
The segments overlap each other; i.e. both include the same address.

25 **Corrupt file. Unable to find module** *module* **(***file***)**
A module is missing. Linker aborts immediately.

26 **Segment** *segment* **is too long**
This error should never occur unless the program is extremely large. Linker aborts immediately.

27 **Entry entry in module** *module* **(***file***) redefined in module** *module* **(***file***)**
There are two or more entries with the same name. Linker aborts immediately.

28 **File** *file* **is too long**
The program is too large. Split the file. Linker aborts immediately.

29 **No object file specified in command-line**
There is nothing to link. Linker aborts immediately.

30 **Option** *option* **also requires the** *option* **option**
Linker aborts immediately.

31 **Option** *option* **cannot be combined with the** *option* **option**
Linker aborts immediately.

32 **Option** *option* **cannot be combined with the** *option* **option and the** *option* **option**
Linker aborts immediately.

33 **Faulty value** *value***, (range is 10-150)**
Faulty page setting. Linker aborts immediately.

34 **Filename too long**
The filename is more than 255 characters long. Linker aborts immediately.

35 **Unknown flag** *flag* **in cross reference option** *option*
Linker aborts immediately.

36 **Option** *option* **does not exist**
Linker aborts immediately.

37 **- not succeeded by character**
The - (dash) marks the beginning of an option, and must be followed by a character. Linker aborts immediately.

38 **Option** *option* **must not be defined more than once**
Linker aborts immediately.

39 **Illegal character specified in option** *option*
Linker aborts immediately.

40 **Argument expected after option** *option*
This option must be succeeded by an argument. Linker aborts immediately.

41 **Unexpected '-' in option** *option*
Linker aborts immediately.

42 **Faulty symbol definition -D***symbol definition*
Incorrect syntax. Linker aborts immediately.

43 **Symbol in symbol definition too long**
The symbol name is more than 255 characters. Linker aborts immediately.

44 **Faulty value** *value***, (range 80-300)**
Faulty column setting. Linker aborts immediately.

45 **Unknown CPU** *CPU* **encountered in** *context*
Linker aborts immediately. Make sure that the argument to `-c` is valid. If you are using the command line you can get a list of CPUs by typing `xlink -c?`.

46 **Undefined external** *external* **referred in** *module* **(***file***)**
Entry to *external* is missing.

47 **Unknown format** *format* **encountered in** *context*
Linker aborts immediately.

48 **This error message number is not used.**

49 **This error message number is not used.**

50 **Paragraph no. not allowed for this CPU, encountered in option** *option*
Linker aborts immediately. Do not use paragraph numbers in declarations.

51 *Input base* **value expected in option** *option*
Linker aborts immediately.

52 **Overflow on value in option** *option*
Linker aborts immediately.

53 **Parameter exceeded 255 characters in extended command line file** *file*
Linker aborts immediately.

54 **Extended command line file** *file* **is empty**
Linker aborts immediately.

55 **Extended command line variable XLINK_ENVPAR is empty**
Linker aborts immediately.

56 **Non-increasing range in segment definition** *segment def*
Linker aborts immediately.

57 **No CPU defined**
No CPU defined, either in the command line or in XLINK_CPU. Linker aborts immediately.

58 **No format defined**
No format defined, either in the command line or in XLINK_FORMAT. Linker aborts immediately.

59 **Revision no. for file is incompatible with XLINK revision no.**
Linker aborts immediately.

If this error occurs after recompilation or reassembly, the wrong version of XLINK is being used. Check with your supplier.

60 **Segment** *segment* **defined in bank definition and segment definition.**
Linker aborts immediately.

61 **This error message number is not used.**

62 **Input file** *file* **cannot be loaded more than once**
Linker aborts immediately.

63 **Trying to pop an empty stack in module** *module* **(***file***)**
Linker aborts immediately. Recompile or reassemble.

64 **Module** *module* **(***file***) has not the same debug type as the other modules**
Linker aborts immediately.

65 **Faulty replacement definition -e *replacement* definition**
Incorrect syntax. Linker aborts immediately.

66 **Function with F-index *index* has not been defined before indirect reference in module *module* (*file*)**
Indirect call to an undefined in module. Probably caused by an omitted function declaration.

67 **Function *name* has same F-index as *function*-*name*, defined in module *module* (*file*)**
Probably a corrupt file. Recompile file.

68 **External function *name* in module *module* (*file*) has no global definition**
If no other errors have been encountered, this error is generated by an assembly-language call from C where the required declaration using the `$DEFFN` assembly-language support directive is missing. The declaration is necessary to inform the linker of the memory requirements of the function.

69 **Indirect or recursive function *name* in module *module* (file) has parameters or auto variables in nondefault memory**
The recursively or indirectly called function name is using extended language memory specifiers (bit, data, idata, etc) to point to non-default memory, memory which is not allowed.

Function parameters to indirectly called functions must be in the default memory area for the memory model in use, and for recursive functions, both local variables and parameters must be in default memory.

70 **This error message number is not used.**

71 **Segment *segment* is incorrectly defined (in a bank definition, has wrong segment type or mixed with other segment types)**
This is usually due to misuse of a predefined segment; see the explanation of *segment* in the *IAR Compiler Reference Guide*. It may be caused by changing the predefined linker control file.

72 **Segment *name* must be defined in a segment option definition (-Z, -b, or -P)**
This is caused either by the omission of a segment in the linker (usually a segment needed by the C system control) file or by a spelling error (segment names are case sensitive).

73 **Label ?ARG_MOVE not found (recursive function needs it)**
In the library there should be a module containing this label. If it has been removed it must be restored.

74 **There was an error when writing to file** *file*
Either the linker or your host system is corrupt, or the two are incompatible.

75 **SFR address in module** *module* **(***file***), segment** *segment* **at address** *address***, value** *value* **is out of bounds**
A special function register (SFR) has been defined to an incorrect address. Change the definition.

76 **Absolute segments overlap in module** *module*
The linker has found two or more absolute segments in *module* overlapping each other.

77 **Absolute segments in module** *module* **(***file***) overlaps absolute segment in module** *module* **(***file***)**
The linker has found two or more absolute segments in *module* (*file*) and *module* (*file*) overlapping each other.

78 **Absolute segment in module** *module* **(***file***) overlaps segment** *segment*
The linker has found an absolute segment in *module* (*file*) overlapping a relocatable segment.

79 **Faulty allocation definition -a** *definition*
The linker has discovered an error in an overlay control definition.

80 **Symbol in allocation definition (-a) too long**
A symbol in the `-a` command is too long.

81 **Unknown flag in extended format option** *option*
Make sure that the flags are valid.

82 **Conflict in segment** *name***. Mixing overlayable and not overlayable segment parts.**
These errors only occur with the 8051 and converted PL/M code.

83 **The overlayable segment** *name* **may not be banked.**
These errors only occur with the 8051 and converted PL/M code.

84 **The overlayable segment** *name* **must be of relative type.**
These errors only occur with the 8051 and converted PL/M code.

85 **The far/farc segment** *name* **in module** *module* **(***file***) is larger than** *size*
The segment *name* is too large to be a `far` segment.

86 **This error message number is not used.**

87 **Function with F-index *i* has not been defined before tiny_func referenced in module *module* (*file*)**
Check that all tiny functions are defined before they are used in a module.

88 **Wrong library used (compiler version or memory model mismatch). Problem found in *module* (*file*). Correct library tag is *tag***
Code from this compiler needs a matching library. A library belonging to a later or earlier version of the compiler may have been used.

89 **This error message can only be encountered with demo products.**

90 **This error message can only be encountered with demo products.**

91 **This error message can only be encountered with demo products.**

92 **Cannot use this format with this CPU**
Some formats need CPU-specific information and are only supported for some CPUs.

93 **Non-existant warning number *number*, (valid numbers are 0-*max*)**
An attempt to suppress a warning that does not exist gives this error.

94 **Unknown flag *x* in local symbols option -n*x***
The character *x* is not a valid flag in the local symbols option.

95 **Module *module* (*file*) uses source file references, which are not available in UBROF 5 output**
This feature cannot be filtered out by the linker when producing UBROF 5 output.

96 **Unmatched -! comment in extended command file**
An odd number of `-!` (comment) options were seen in a linker command file.

97 **Unmatched -! comment in extended command line variable XLINK_ENVPAR**
As above, but for the environment variable `XLINK_ENVPAR`.

98 **Unmatched /* comment in extended command file**
No matching `*/` was found in the linker command file.

99 **Syntax error in segment definition:** *option*
There was a syntax error in the option.

100 **Segment name too long:** *segment* **in** *option*
The segment name exceeds the maximum length (255 characters).

101 **Segment already defined:** *segment* **in** *option*
The segment has already been mentioned in a segment definition option.

102 **No such segment type:** *option*
The specified segment type is not valid.

103 **Ranges must be closed in** *option*
The `-P` option requires all memory ranges to have an end.

104 **Failed to fit all segments into specified ranges. Problem discovered in segment** *segment***.**
The packing algorithm used in the linker did not manage to fit all the segments.

105 **Recursion not allowed for this system. Check module map for recursive functions**
The run-time model used does not support recursion. Each function determined by the linker to be recursive is marked as such in the module map part of the linker list file.

106 **Syntax error or bad argument in** *option*
There was an error when parsing the command line argument given.

107 **Banked segments do not fit into the number of banks specified**
The linker did not manage to fit all of the contents of the banked segments into the banks given.

108 **Cannot find function** *function* **mentioned in -a#**
All the functions specified in an indirect call option must exist in the linked program.

109 **Function** *function* **mentioned as callee in -a# is not indirectly called**
Only functions that actually can be called indirectly can be specified to do so in an indirect call option.

110 **Function** *function* **mentioned as caller in -a# does not make indirect calls**
Only functions that actually make indirect calls can be specified to do so in an indirect call option.

111 **The file** *file* **is not a UBROF file**
The contents of the file are not in a format that XLINK can read.

112 **The module** *module* **is for an unknown CPU (tid =** *tid***). Either the file is corrupt or you need a later version of XLINK**
The version of XLINK used has no knowledge of the CPU that the file was compiled/assembled for.

113 **Corrupt input file:** *symptom* **in module** *module* **(***file* **)**
The input file indicated appears to be corrupt. This can occur either because the file has for some reason been corrupted after it was created, or because of a problem in the compiler/assembler used to create it. If the latter appears to be the case, please contact IAR Technical Support.

114 **This error message number is not used.**

115 **Unmatched '"' in extended command file or XLINK_ENVPAR**
When parsing an extended command file or the environment variable XLINK_ENVPAR, XLINK found an unmatched quote character.

For filenames with quote characters you need to put a backslash before the quote character. For example, writing

```
c:\iar\"A file called \"file\""
```

will cause XLINK to look for a file called

```
A file called "file"
```

in the `c:\iar\directory`.

116 **Definition of** *symbol* **in module** *module1* **is not compatible with definition of** *symbol* **in module** *module2*
The symbol *symbol* has been tentatively defined in one or both of the modules. Tentative definitions must match other definitions.

117 **Incompatible runtime modules. Module** *module1* **specifies that** *attribute* **must be** *value1***, but module** *module2* **has the value** *value2*
These modules cannot be linked together. They were compiled with settings that resulted in incompatible run-time modules.

118 **Incompatible runtime modules. Module** *module1* **specifies that** *attribute* **must be** *value***, but module** *module2* **specifies no value for this attribute.**
These modules cannot be linked together. They were compiled with settings that resulted in incompatible run-time modules.

119 **Cannot handle C++ identifiers in this output format**
The selected output format does not support the use of C++ identifiers (block-scoped names or names of C++ functions).

120 **Overlapping address ranges for address translation.** *address type* **address** *address* **is in more than one range**
The address *address* (of logical or physical type) is the source or target of more than one address translation command.

If, for example, both `-M0-2FFF=1000` and `-M2000-3FFF=8000` are given, this error may be given for any of the logical addresses in the range `2000-2FFF`, for which to separate translation commands have been given.

121 Segment part or absolute content at logical addresses *`start`*`-`*`end`* would be translated into more than one physical address range

The current implementation of address translation does not allow logical addresses from one segment part (or the corresponding range for absolute parts from assembler code) to end up in more than one physical address range.

If, for example, `-M0-1FFF=10000` and `-M2000-2FFF=20000` are used, a single segment part is not allowed to straddle the boundary at address `2000`.

122 The address *`address`* is too large to be represented in the output format *`format`*

The selected output format *`format`* cannot represent the address *`address`*. For example, the output format `INTEL-STANDARD` can only represent addresses in the range `0-FFFF`.

123 The output format *`format`* does not support address translation (`-M`, `-b#`, or `-b@`)

Address translation is not supported for all output formats.

124 Segment conflict for segment *`segment`*. In module *`module1`* there is a segment part that is of type *`type1`*, while in module *`module2`* there is a segment part that is of type *`type2`*

All segment parts for a given segment must be of the same type. One reason for this conflict can be that a `COMMON` segment is mistakenly declared `RSEG` (relocatable) in one module.

125 This error message number is not used.

126 Runtime model attribute "__cpu" not found. Please enter at least one line in your assembly code that contains the following statement: RTMODEL "__cpu", "16C61". Replace 16C61 with your chosen CPU. The CPU must be in uppercase.

The `__cpu` runtime model attribute is needed when producing COFF output. The compiler always supplies this attribute, so this error can only occur for programs consisting entirely of assembler modules.

At least one of the assembler modules must supply this attribute.

127 Segment placement command "*command*" provides no address range, but the last address range(s) given is the wrong kind (bit addresses versus byte addresses).
This error will occur if something like this is entered:

```
-Z(DATA)SEG=1000-1FFF
-Z(BIT)BITVARS=
```

Note that the first uses byte addresses and the second needs bit addresses. To avoid this, provide address ranges for both.

128 Segments cannot be mentioned more than once in a copy init command: "-Q*args*"
Each segment must be either the source or the target of a copy init command.

129 This error message number is not used.

130 Segment placement needs an address range: "*command*"
The first segment placement command (`-Z`, `-P`) must have an address range.

131 Far segment type illegal in packed placement command: "*command*". Use explicit address intervals instead. For example: `[20000-4FFFF]/10000`
Using a far segment type (`FARCODE`, `FARDATA`, `FARCONST`) is illegal in packed placement (`-P`).

132 Module *module* (*file*) uses UBROF version 9.0. This version of UBROF was temporary and is no longer supported by XLINK
Support for UBROF 9.0.0 has been dropped from XLINK starting with XLINK 4.53A.

133 The output format *format* cannot handle multiple address spaces. Use format variants (`-y -O`) to specify which address space is wanted
The output format used has no way to specify an address space. The format variant modifier used can be prefixed with a segment type to restrict output to the corresponding address space only. For example, `-Fmotorola -y(CODE)` will restrict output to bytes from the address space used for the `CODE` segment type.

See *Restricting the output to a single address space*, page 47 for more information.

Warning messages

The following section lists the linker warning messages:

0 **Too many warnings**
Too many warnings encountered.

1 **Error tag encountered in module** *module* **(***file***)**
A UBROF error tag was encountered when loading file *file*. This indicates a corrupt file and will generate an error in the linking phase.

2 **Symbol** *symbol* **is redefined in command-line**
A symbol has been redefined.

3 **Type conflict. Segment** *segment***, in module** *module***, is incompatible with earlier segment(s) of the same name**
Segments of the same name should have the same type.

4 **Close/open conflict. Segment** *segment***, in module** *module***, is incompatible with earlier segment of the same name**
Segments of the same name should be either open or closed.

5 **Segment** *segment* **cannot be combined with previous segment**
The segments will not be combined.

6 **Type conflict for external/entry** *entry***, in module** *module***, against external/entry in module** *module*
Entries and their corresponding externals should have the same type.

7 **Module** *module* **declared twice, once as program and once as library. Redeclared in file** *file***, ignoring library module**
The program module is linked.

8 **This warning message number is not used.**

9 **Ignoring redeclared program entry in module** *module* **(***file***), using entry from module** *module1*
Only the program entry found first is chosen.

10 **No modules to link**
The linker has no modules to link.

11 **Module** *module* **declared twice as library. Redeclared in file** *file***, ignoring second module**
The module found first is linked.

12 **Using SFB in banked segment** *segment* **in module** *module* **(***file***)**
The `SFB` assembler directive may not work in a banked segment.

13 **Using SFE in banked segment** *segment* **in module** *module* **(***file***)**
The SFE assembler directive may not work in a banked segment.

14 **Entry** *entry* **duplicated. Module** *module* **(***file***) loaded, module** *module* **(***file***) discarded**
Duplicated entries exist in conditionally loaded modules; i.e. library modules or conditionally loaded program modules (with the -C option).

15 **Predefined type sizing mismatch between modules** *module* **(***file***) and** *module* **(***file***)**
The modules have been compiled with different options for predefined types, such as different sizes of basic C types (e.g. integer, double).

16 **Function** *name* **in module** *module* **(***file***) is called from two function trees (with roots** *name1* **and** *name2***)**
The probable cause is *module* interrupt function calls another function that also could be executed by a foreground program, and this could lead to execution errors.

17 **Segment name is too large or placed at wrong address**
This error occurs if a given segment overruns the available address space in the named memory area. To find out the extent of the overrun do a dummy link, moving the start address of the named segment to the lowest address, and look at the linker map file. Then relink with the correct address specification.

18 **Segment** *segment* **overlaps segment** *segment*
The linker has found two relocatable segments overlapping each other. Check the segment placement option parameters.

19 **Absolute segments overlaps in module** *module* **(***file***)**
The linker has found two or more absolute segments in module *module* overlapping each other.

20 **Absolute segment in module** *module* **(***file***) overlaps absolute segment in module** *module* **(***file***)**
The linker has found two or more absolute segments in module *module* (*file*) and module *module* (*file*) overlapping each other. Change the ORG directives.

21 **Absolute segment in module** *module* **(***file***) overlaps segment** *segment*
The linker has found an absolute segment in module *module* (*file*) overlapping a relocatable segment. Change either the ORG directive or the -Z relocation command.

22 **Interrupt function *name* in module *module* (*file*) is called from other functions**
Interrupt functions may not be called.

23 *limitation-specific warning*
Due to some limitation in the chosen output format, or in the information available, XLINK cannot produce the correct output. Only one warning for each specific limitation is given.

24 *num* **counts of** *warning* **total**
For each warning of type 23 emitted, a summary is provided at the end.

25 **Using -Y# discards and distorts debug information. Use with care. If possible find an updated debugger that can read modern UBROF**
Using the UBROF format modifier `-Y#` is not recommended.

26 **No reset vector found**
Failed in determining the `LOCATION` setting for XCOFF output format for the 78400 processor, because no reset vector was found.

27 **No code at the start address**
Failed in determining the `LOCATION` setting for XCOFF output format for the 78400 processor, because no code was found at the address specified in the reset vector.

28 **Parts of segment *name* are initialized, parts not**
Segments should not be partially initialized and partially uninitialized, if the result of the linking is meant to be promable.

29 **Parts of segment *name* are initialized, even though it is of type *type* (and thus not promable)**
`DATA` memory should not be initialized if the result of the linking is meant to be promable.

30 **Module *name* is compiled with tools for *cpu1* expected *cpu2***
You are building an executable for CPU *cpu2*, but module *name* is compiled for CPU *cpu1*.

31 **Modules have been compiled with possibly incompatible settings: *more information***
According to the contents of the modules, they are not compatible.

32 **Format option set more than once. Using format *format***
The format option can only be given once. The linker uses the format *format*.

33 **Using -r overrides format option. Using UBROF**
The `-r` option specifies UBROF format and C-SPY™ library modules. It overrides any `-F` (format) option.

34 **The 20 bit segmented variant of the INTEL EXTENDED format cannot represent the addresses specified. Consider using -Y1 (32 bit linear addressing).**
The program uses addresses higher than `0xFFFFF`, and the segmented variant of the chosen format cannot handle this. The linear-addressing variant can handle full 32-bit addresses.

35 **There is more than one definition for the struct/union type with tag** *`tag`*
Two or more different structure/union types with the same tag exist in the program. If this is not intentional, it is likely that the declarations differ slightly. It is very likely that there will also be one or more warnings about type conflicts (warning 6). If this is intentional, consider turning this warning off.

36 **There are indirectly called functions doing indirect calls. This can make the static overlay system unreliable**
XLINK does not know what functions can call what functions in this case, which means that it cannot make sure static overlays are safe.

37 **More than one interrupt function makes indirect calls. This can make the static overlay system unreliable. Using -ai will avoid this**
If a function is called from an interrupt while it is already running, its params and locals will be overwritten.

38 **There are indirect calls both from interrupts and from the main program. This can make the static overlay system unreliable. Using -ai will avoid this**
If a function is called from an interrupt while it is already running, its params and locals will be overwritten.

39 **The function** *`function`* **in module** *`module`* **(***`file`***) does not appear to be called. No static overlay area will be allocated for its params and locals**
As far as XLINK can tell, there are no callers for the function, so no space is needed for its params and locals. To make XLINK allocate space anyway, use `-a(`*`function`*`)`.

40 **The module** *`module`* **contains obsolete type information that will not be checked by the linker**
This kind of type information is no longer used.

41 **The function *function* in module *module* (*file*) makes indirect calls but is not mentioned in the left part of any -a# declaration**
If any `-a#` indirect call options are given they must, taken together, specify the complete picture.

42 **This warning message number is not used.**

43 **The function *function* in module *module* (*file*) is indirectly called but is not mentioned in the right part of any -a# declaration**
If any `-a#` indirect call options are given they must, taken together, specify the complete picture.

44 **C library routine localtime failed. Timestamps will be wrong**
XLINK is unable to determine the correct time. This primarily affects the dates in the list file. This problem has been observed on one host platform if the date is after the year 2038.

45 **Memory attribute info mismatch between modules *module1* (*file1*) and *module2* (*file2*)**
The UBROF 7 memory attribute information in the given modules is not the same.

46 **External function *function* in module *module* (*file*) has no global definition**
This warning replaces error 68.

47 **Range error in module *module* (*file*), segment *segment* at address *address*. Value *value*, in tag *tag*, is out of bounds *bounds***
This replaces error 18 when `-Rw` is specified.

48 **Corrupt input file: *symptom* in module *module* (*file*)**
The input file indicated appears to be corrupt. This warning is used in preference to error 113 when the problem is not serious, and is unlikely to cause trouble.

49 **Using SFB/SFE in module *module* (*file*) for segment *segment*, which has no included segment parts**
SFB/SFE (assembler directives for getting the start or end of a segment) has been used on a segment for which no segment parts were included.

50 **There was a problem when trying to embed the source file *source* in the object file**
This warning is given if the file *source* could not be found or if there was an error reading from it. XLINK searches for source files in the same places as it searches for object files, so including the directory where the source file is located in the XLINK **Include** (`-I`) option could solve the problem.

51 **Some source reference debug info was lost when translating to UBROF 5 (example: statements in *function* in module *module*)**
UBROF 6 file references can handle source code in more than one source file for a module. This is not possible in UBROF 5 embedded source, so any references to files not included have been removed.

52 **More than one definition for the byte at address *address* in common segment *segment***
The most probable cause is that more than one module defines the same interrupt vector.

53 **Some untranslated addresses overlap translation ranges. Example: Address *addr1* (untranslated) conflicts with logical address *addr2* (translated to *addr1*)**
This can be caused by something like this:

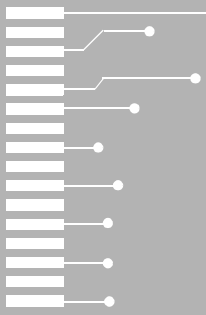
```
-Z(CODE)SEG1=1000-1FFF
-Z(CODE)SEG2=2000-2FFF
-M(CODE)1000=2000
```

This will place `SEG1` at logical address `1000` and `SEG2` at logical address `2000`. However, the translation of logical address `1000` to physical address `2000` and the absence of any translation for logical address `1000` will mean that in the output file, both `SEG1` and `SEG2` will appear at physical address `1000`.

54 **This warning message has not been implemented yet.**

55 **No source level debug information will be generated for modules using the UBROF object format version 8 or earlier. One such module is *module* (*file*)**
When generating UBROF 9 output, essential debug information is not present in input files using UBROF 8 or earlier. For these files all debug information will be suppressed in the output file.

56 **A long filename may cause MPLAB to fail to display the source file: '*pathname*'**
When outputting `COFF` output for the PIC and PIC18 processors on a Windows host, the output file contains a reference to a source file that needs long filenames in order to work. MPLAB cannot handle long filenames.



Part 2: The IAR Library Tools

This part of the IAR Linker and Library Tools Reference Guide contains the following chapters:

- Introduction to the IAR library tools
- XAR
- XAR diagnostics
- XLIB options
- XLIB diagnostics.

Introduction to the IAR library tools

This chapter describes XAR Library Builder™ and IAR XLIB Librarian™—the IAR library tools that enable you to manipulate the relocatable object files produced by the IAR Systems assembler and compiler.

Both tools use the UBROF standard object format (Universal Binary Relocatable Object Format).

Libraries

A library is a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file as it is needed.

Often, modules in a library file have the `LIBRARY` attribute, which means that they will only be loaded by the linker if they are actually needed in the program. This is referred to as *demand loading* of modules.

On the other hand, a module with the `PROGRAM` attribute is *always* loaded when the file in which it is contained is processed by the linker.

A library file is no different from any other relocatable object file produced by the assembler or compiler, it can include modules of both the `LIBRARY` and the `PROGRAM` type.

IAR XAR Library Builder and IAR XLIB Librarian

There are two library tools included with your IAR Systems product. The first of them, IAR XAR Library Builder can only do one thing: combine a set of UBROF object files into a library file. IAR XLIB Librarian, on the other hand, can do a number of things in addition to building libraries: modify the size and contents of existing libraries, list information about individual library modules, and more.

Note: XAR does not distinguish between UBROF versions for different processors. It is up to you to make sure that you are not building a library consisting of files from different CPUs.

Also note that XAR allows you to specify the same object file twice or even more times. Make sure to avoid this, as the result would be a library file with multiply defined contents.

CHOOSING WHICH TOOL TO USE

Whether you should use XAR or XLIB depends on what you want to achieve, and on the complexity of your project. If all you need to do is to combine a number of source object files into a library file, XAR is enough for your purposes, and simpler to use than XLIB. However, if you need to modify a library or the modules it consists of, you must use XLIB.

Using libraries with C/Embedded C++ programs

All C/Embedded C++ programs make use of libraries, and the IAR Systems compilers are supplied with a number of standard library files.

Most C/Embedded C++ programmers will use one or both of the IAR library tools at some point, for one of the following reasons:

- To replace or modify a module in one of the standard libraries. For example, XLIB can be used for replacing the distribution versions of the `CSTARTUP` and/or `putchar` modules with ones that you have customized.
- To add C/Embedded C++ or assembler modules to the standard library file so they will always be available whenever a C/Embedded C++ program is linked. You use XLIB for this.
- To create custom library files that can be linked into their programs, as needed, along with the standard C/Embedded C++ library. You can use both XAR and XLIB for this.

Using libraries with assembler programs

If you are only using assembler you do not *need* to use libraries. However, libraries provide the following advantages, especially when writing medium- and large-sized assembler applications:

- They allow you to combine utility modules used in more than one project into a simple library file. This simplifies the linking process by eliminating the need to include a list of input files for all the modules you need. Only the library module(s) needed for the program will be included in the output file.
- They simplify program maintenance by allowing multiple modules to be placed in a single assembler source file. Each of the modules can be loaded independently as a library module.
- They reduce the number of object files that make up an application, maintenance, and documentation.

You can create your assembly language library files using one of two basic methods:

- A library file can be created by assembling a single assembler source file which contains multiple library-type modules. The resulting library file can then be modified using XLIB.
- A library file can be produced by using XAR or XLIB to merge any number of existing modules together to form a user-created library.

The `NAME` and `MODULE` assembler directives are used for declaring modules as being of `PROGRAM` or `LIBRARY` type, respectively.

For additional information, see the *IAR Assembler Reference Guide*.

XAR

This chapter describes how to use the IAR XAR Library Builder™.

Using XAR

XAR is run from the command line, using the command `xar`.

BASIC SYNTAX

If you run the IAR XAR Library Builder without giving any command line options, the default syntax is:

```
xar libraryfile objectfile1 ... objectfileN
```

Parameters

The parameters are:

Parameter	Description
libraryfile	The file to which the module(s) in the object file(s) will be sent.
objectfile1 ... objectfileN	The object file(s) containing the module(s) to build the library from.

Table 24: XAR parameters

Example

The following example creates a library file called `mylibrary.r19` from the source object files `module1.r19`, `module2.r19`, and `module3.r19`:

```
xar mylibrary.r19 module1.r19 module2.r19 module3.r19
```

Summary of XAR options

The following table shows a summary of the XAR options:

Option	Description
`-o`	Specifies the library file.
`-V`	Provides user feedback.

Table 25: XAR options summary

Descriptions of XAR options

The following sections give detailed reference information for each XAR option.

`-o` `-o` *`libraryfile`*

By default, XAR assumes the first argument after the `xar` command to be the name of the destination library file. Use the `-o` option if you want to specify the library file you are creating elsewhere on the command line instead.

Example

The following example creates a library file called `mylibrary.r19` from the source modules `module1.r19`, `module2.r19`, and `module3.r19`:

```
xar module1.r19 module2.r19 module3.r19 -o mylibrary.r19
```

`-V` `-V`

When this command is used, XAR reports which operations it performs, in addition to giving diagnostic messages. This is the default setting when running XAR from the IAR Embedded Workbench™.

XAR diagnostics

This chapter lists the messages produced by the IAR XAR Library Builder™.

XAR messages

The following section lists the XAR messages.

0 **Not enough memory**
XAR was unable to aquire the memory that it needed.

1 **-o option requires an argument**
XAR expects an argument after `-o`.

2 **Unknown option** `option`
XAR encountered an unknown option on the command line.

3 **Too few arguments**
XAR expects to find more arguments

4 **Same file as both input and output:** `filename`
One of the files is used as both source object file and destination library. This is illegal since it would overwrite the source object file. If you want to give the new library a name that is used by one of the source object files, you must use a temporary filename for the library you are building with XAR and rename that temporary file afterwards.

5 **Can't open library file** `filename` **for writing**
XAR was unable to open the library file for writing. Make sure that the library file is not write protected.

6 **Can't open object file** `filename`
XAR was unable to open the object file. Make sure that the file exists.

7 **Error occured while writing to library file**
An error ocurred while XAR was writing to the file.

8 `filename` **is not a valid UBROF file**
The file is not a valid UBROF file.

9 **Error occured while reading from** `filename`
An error ocurred while XAR was reading the file.

10 **Error occured while closing** `filename`
An error ocurred while XAR was closing the file.

11 **XAR didn't find any bytes to read in** `filename`
The object file seems to be empty.

12 `filename` **didn't end as a valid UBROF file should**
The file did not end as a UBROF file is supposed to end. Either the file is corrupt or the assembler/compiler produces corrupt output.

13 **XAR can't** `fseek` **in library file**
The call to `fseek` failed.

XLIB options

This chapter summarizes the IAR XLIB Librarian™ options, classified according to their function, and gives a detailed syntactic and functional description of each XLIB option.

Using XLIB options

XLIB can be run from the command line or from a batch file.

GIVING XLIB OPTIONS FROM THE COMMAND LINE

The `-c` command line option allows you to run XLIB options from the command line. Each argument specified after the `-c` option is treated as one XLIB option.

For example, specifying:

```
xlib -c "LIST-MOD math.rnn" "LIST-MOD mod.rnn m.txt"
```

is equivalent to entering the following options in XLIB:

```
*LIST-MOD math.rnn
*LIST-MOD mod.rnn m.txt
*QUIT
```

Note: Each command line argument must be enclosed in double quotes if it includes spaces.

The individual words of an identifier can be abbreviated to the limit of ambiguity. For example, `LIST-MODULES` can be abbreviated to `L-M`.

When running XLIB you can press Enter at any time to prompt for information, or display a list of the possible options.

XLIB BATCH FILES

Running XLIB with a single command-line parameter specifying a file, causes XLIB to read options from that file instead of from the console.

PARAMETERS

The following parameters are common to many of the XLIB options.

Parameter	What it means	
objectfile	File containing object modules.	
start, end	The first and last modules to be processed, in one of the following forms:	
	n	The *n*th module.
	$	The last module.
	name	Module *name*.
	name+n	The module *n* modules after *name*.
	$-*n*	The module *n* modules before the last.
listfile	File to which a listing will be sent.	
source	A file from which modules will be read.	
destination	The file to which modules will be sent.	

Table 26: XLIB parameters

MODULE EXPRESSIONS

In most of the XLIB options you can or must specify a source module (like *oldname* in RENAME-MODULE), or a range of modules (*startmodule, endmodule*).

Internally in all XLIB operations, modules are numbered from 1 in ascending order. Modules may be referred to by the actual name of the module, by the name plus or minus a relative expression, or by an absolute number. The latter is very useful when a module name is very long, unknown, or contains unusual characters such as space or comma.

The following table shows the available variations on module expressions:

Name	Description
3	The third module.
$	The last module.
name+4	The module 4 modules after *name*.
name-12	The module 12 modules before *name*.
$-2	The module 2 modules before the last module.

Table 27: XLIB module expressions

The option LIST-MOD FILE,,$-2 will thus list the three last modules in FILE on the terminal.

LIST FORMAT

The `LIST` options give a list of symbols, where each symbol has one of the following prefixes:

Prefix	Description
nn`.Pgm`	A program module with relative number *nn*.
nn`.Lib`	A library module with relative number *nn*.
`Ext`	An external in the current module.
`Ent`	An entry in the current module.
`Loc`	A local in the current module.
`Rel`	A standard segment in the current module.
`Stk`	A stack segment in the current module.
`Com`	A common segment in the current module.

Table 28: XLIB list option symbols

USING ENVIRONMENT VARIABLES

The IAR XLIB Librarian™ supports a number of environment variables. These can be used for creating defaults for various XLIB options so that they do not have to be specified on the command line.

The following environment variables can be used by XLIB:

Environment variable	Description
`XLIB_COLUMNS`	Sets the number of list file columns in the range 80–132. The default is 80. For example, to set the number of columns to 132: `set XLIB_COLUMNS=132`
`XLIB_CPU`	Sets the CPU type so that the `DEFINE-CPU` option will not be required when you start an XLIB session. For example, to set the CPU type to *chipname*: `set XLIB_CPU=`*`chipname`*

Table 29: XLIB environment variables

Environment variable	Description
XLIB_PAGE	Sets the number of lines per list file page in the range 10–100. The default is a listing without page breaks. For example, to set the number of lines per page to 66: `set XLIB_PAGE=66`
XLIB_SCROLL_BREAK	Sets the scroll pause in number of lines to make the XLIB output pause and wait for the Enter key to be pressed after the specified number of lines (16–100) on the screen have scrolled by. For example, to pause every 22 lines: `set XLIB_SCROLL_BREAK=22`

Table 29: XLIB environment variables (Continued)

Summary of XLIB options for all UBROF versions

The following table shows a summary of the XLIB options:

Option	Description
COMPACT-FILE	Shrinks library file size.
DEFINE-CPU	Specifies CPU type.
DELETE-MODULES	Removes modules from a library.
DIRECTORY	Displays available object files.
DISPLAY-OPTIONS	Displays XLIB options.
ECHO-INPUT	Command file diagnostic tool.
EXIT	Returns to operating system.
FETCH-MODULES	Adds modules to a library.
HELP	Displays help information.
INSERT-MODULES	Moves modules in a library.
LIST-ALL-SYMBOLS	Lists every symbol in modules.
LIST-CRC	Lists CRC values of modules.
LIST-DATE-STAMPS	Lists dates of modules.
LIST-ENTRIES	Lists PUBLIC symbols in modules.
LIST-EXTERNALS	Lists EXTERN symbols in modules.
LIST-MODULES	Lists modules.

Table 30: XLIB options summary

Option	Description
LIST-OBJECT-CODE	Lists low-level relocatable code.
LIST-SEGMENTS	Lists segments in modules.
MAKE-LIBRARY	Changes a module to library type.
MAKE-PROGRAM	Changes a module to program type.
ON-ERROR-EXIT	Quits on a batch error.
QUIT	Returns to operating system.
REMARK	Comment in command file.
REPLACE-MODULES	Updates executable code.

Table 30: XLIB options summary (Continued)

Note: There are some XLIB options that do not work with the output from modern IAR C/EC++ compilers or assemblers. See *Summary of XLIB options for older UBROF versions*, page 96.

Descriptions of XLIB options for all UBROF versions

The following section gives detailed reference information for each option.

COMPACT-FILE

COMPACT-FILE *objectfile*

Use COMPACT-FILE to reduce the size of the library file by concatenating short, absolute records into longer records of variable length. This will decrease the size of a library file by about 5%, in order to give library files which take up less time during the loader/linker process.

Example

The following option compacts the file maxmin.*rnn*:

```
COMPACT-FILE maxmin
```

This displays:

```
20 byte(s) deleted
```

DEFINE-CPU

DEFINE-CPU *cpu*

Use this option to specify the CPU type *cpu*. This option must be issued before any operations on object files can be done.

Examples

The following option defines the CPU as *chipname*:

```
DEF-CPU chipname
```

DELETE-MODULES

```
DELETE-MODULES objectfile start end
```

Use `DELETE-MODULES` to remove the specified modules from a library.

Examples

The following option deletes module `2` from the file `math.rnn`:

```
DEL-MOD math 2 2
```

DIRECTORY

```
DIRECTORY [specifier]
```

Use `DIRECTORY` to display on the terminal all available object files of the type that applies to the target processor. If no *`specifier`* is given, the current directory is listed.

Examples

The following option lists object files in the current directory:

```
DIR
```

It displays:

```
general      770
math         502
maxmin       375
```

DISPLAY-OPTIONS

```
DISPLAY-OPTIONS [listfile]
```

Displays XLIB options.

Use `DISPLAY-OPTIONS` to list in the *`listfile`* the names of all the CPUs which are recognized by this version of the IAR XLIB Librarian. After that a list of all UBROF tags is output.

Examples

To list the options to the file `opts.lst`:

```
DISPLAY-OPTIONS opts
```

ECHO-INPUT `ECHO-INPUT`

`ECHO-INPUT` is a command file diagnostic tool which you may find useful when debugging command files in batch mode as it makes all command input visible on the terminal. In the interactive mode it has no effect.

Examples

In a batch file

```
ECHO-INPUT
```

echoes all subsequent XLIB options.

EXIT `EXIT`

Use `EXIT` to exit from XLIB after an interactive session and return to the operating system.

Examples

To exit from XLIB:

```
EXIT
```

EXTENSION `EXTENSION` *`extension`*

Use `EXTENSION` to set the default file extension.

FETCH-MODULES `FETCH-MODULES` *`source destination`* [*`start`*] [*`end`*]

Use `FETCH-MODULES` to add the specified modules to the *`destination`* library file. If *`destination`* already exists, it must be empty or contain valid object modules; otherwise it will be created.

Examples

The following option copies the module `mean` from `math.`*`rnn`* to `general.`*`rnn`*:

```
FETCH-MOD math general mean
```

HELP

```
HELP [option] [listfile]
```

Parameters

option	Option for which help is displayed.

Use this option to display help information.

If the HELP option is given with no parameters, a list of the available options will be displayed on the terminal. If a parameter is specified, all options which match the parameter will be displayed with a brief explanation of their syntax and function. A * matches all options. HELP output can be directed to any file.

Examples

For example, the option:

```
HELP LIST-MOD
```

displays:

```
LIST-MODULES <Object file> [<List file>] [<Start module>]
[<End module>]
    List the module names from [<Start module>] to
    [<End module>].
```

INSERT-MODULES

```
INSERT-MODULES objectfile start end {BEFORE | AFTER} dest
```

Use INSERT-MODULES to insert the specified modules in a library, before or after the *dest*.

Examples

The following option moves the module mean before the module min in the file math.*rnn*:

```
INSERT-MOD math mean mean BEFORE min
```

LIST-ALL-SYMBOLS

```
LIST-ALL-SYMBOLS objectfile [listfile] [start] [end]
```

Use LIST-ALL-SYMBOLS to list all symbols (module names, segments, externals, entries, and locals) for the specified modules in the *objectfile*. The symbols are listed to the *listfile*.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists all the symbols in `math.rnn`:

```
LIST-ALL-SYMBOLS math
```

This displays:

```
1.  Lib  max
      Rel   CODE
      Ent   max
      Loc   A
      Loc   B
      Loc   C
      Loc   ncarry
2.  Lib  mean
      Rel   DATA
      Rel   CODE
      Ext   max
      Loc   A
      Loc   B
      Loc   C
      Loc   main
      Loc   start
3.  Lib  min
      Rel   CODE
      Ent   min
      Loc   carry
```

`LIST-CRC` `LIST-CRC` *objectfile* [*listfile*] [*start*] [*end*]

Use `LIST-CRC` to list the module names and their associated CRC values of the specified modules.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists the CRCs for all modules in `math.rnn`:

```
LIST-CRC math
```

This displays:

```
EC41            1.  Lib  max
ED72            2.  Lib  mean
9A73            3.  Lib  min
```

LIST-DATE-STAMPS

```
LIST-DATE-STAMPS objectfile [listfile] [start] [end]
```

Use `LIST-DATE-STAMPS` to list the module names and their associated generation dates for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists the date stamps for all the modules in `math.rnn`:

```
LIST-DATE-STAMPS math
```

This displays:

```
15/Feb/98       1.  Lib  max
15/Feb/98       2.  Lib  mean
15/Feb/98       3.  Lib  min
```

LIST-ENTRIES

```
LIST-ENTRIES objectfile [listfile] [start] [end]
```

Use `LIST-ENTRIES` to list the names and associated entries (`PUBLIC` symbols) for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists the entries for all the modules in `math.rnn`:

```
LIST-ENTRIES math
```

This displays:

```
1.  Lib  max
      Ent   max
2.  Lib  mean
3.  Lib  min
      Ent   min
```

LIST-EXTERNALS

```
LIST-EXTERNALS objectfile [listfile] [start] [end]
```

Use `LIST-EXTERNALS` to list the module names and associated externals (`EXTERN` symbols) for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists the externals for all the modules in `math.rnn`:

```
LIST-EXT math
```

This displays:

```
1.  Lib  max
2.  Lib  mean
      Ext   max
3.  Lib  min
```

LIST-MODULES `LIST-MODULES objectfile [listfile] [start] [end]`

Use `LIST-MODULES` to list the module names for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists all the modules in `math.rnn`:

```
LIST-MOD math
```

It produces the following output:

```
1.  Lib  max
2.  Lib  min
3.  Lib  mean
```

LIST-OBJECT-CODE `LIST-OBJECT-CODE objectfile [listfile]`

Lists low-level relocatable code.

Use `LIST-OBJECT-CODE` to list the contents of the object file on the list file in ASCII format.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists the object code of `math.rnn` to `object.lst`:

```
LIST-OBJECT-CODE math object
```

LIST-SEGMENTS

`LIST-SEGMENTS objectfile [listfile] [start] [end]`

Use `LIST-SEGMENTS` to list the module names and associated segments for the specified modules.

Each symbol is identified with a prefix; see *List format*, page 85.

Examples

The following option lists the segments in the module `mean` in the file `math.rnn`:

```
LIST-SEG math,,mean mean
```

Notice the use of two commas to skip the *listfile* parameter.

This produces the following output:

```
          2.  Lib  mean
                Rel   DATA
                Rel   CODE
```

MAKE-LIBRARY

`MAKE-LIBRARY objectfile [start] [end]`

Changes a module to library type.

Use `MAKE-LIBRARY` to change the module header attributes to conditionally loaded for the specified modules.

Examples

The following option converts all the modules in `main.rnn` to library modules:

```
MAKE-LIB main
```

MAKE-PROGRAM

`MAKE-PROGRAM objectfile [start] [end]`

Changes a module to program type.

Use `MAKE-PROGRAM` to change the module header attributes to unconditionally loaded for the specified modules.

Examples

The following option converts module `start` in `main.rnn` into a program module:

```
MAKE-PROG main start
```

ON-ERROR-EXIT `ON-ERROR-EXIT`

Use `ON-ERROR-EXIT` to make the librarian abort if an error is found. It is suited for use in batch mode.

Examples

The following batch file aborts if the `FETCH-MODULES` option fails:

```
ON-ERROR-EXIT
FETCH-MODULES math new
```

QUIT `QUIT`

Use `QUIT` to exit and return to the operating system.

Examples

To quit from XLIB:

```
QUIT
```

REMARK `REMARK` *`text`*

Use `REMARK` to include a comment in an XLIB command file.

Examples

The following example illustrates the use of a comment in an XLIB command file:

```
REM Now compact file
COMPACT-FILE math
```

REPLACE-MODULES `REPLACE-MODULES` *`source destination`*

Use `REPLACE-MODULES` to update executable code by replacing modules with the same name from *`source`* to *`destination`*. All replacements are logged on the terminal. The main application for this option is to update large run-time libraries etc.

Examples

The following example replaces modules in `math.`*`rnn`* with modules from `newmath.`*`rnn`*:

```
REPLACE-MOD newmath math
```

This displays:

```
Replacing module 'max'
Replacing module 'mean'
Replacing module 'min'
```

Summary of XLIB options for older UBROF versions

There are some XLIB options that do not work with output from IAR C/EC++ compilers or assemblers that output object files in UBROF 8 format and later. This means that these options cannot be used together with compiler/assembler versions delivered with the IAR Embedded Workbench version 3.0 and later, and a few products that were released just before version 3.0. The following table shows a summary of these XLIB options:

Option	Description
RENAME-ENTRY	Renames PUBLIC symbols.
RENAME-EXTERNAL	Renames EXTERN symbols.
RENAME-GLOBAL	Renames EXTERN and PUBLIC symbols.
RENAME-MODULE	Renames one or more modules.
RENAME-SEGMENT	Renames one or more segments.

Table 31: Summary of XLIB options for older compilers

Descriptions of XLIB options for older UBROF versions

The following section gives detailed reference information for each option.

RENAME-ENTRY

RENAME-ENTRY *objectfile old new* [*start*] [*end*]

Use RENAME-ENTRY to rename all occurrences of a PUBLIC symbol from *old* to *new* in the specified modules.

Examples

The following option renames the entry for modules 2 to 4 in math.*rnn* from mean to average:

```
RENAME-ENTRY math mean average 2 4
```

Note: This option does not work with the output from modern IAR C/EC++ compilers or assemblers that produce UBROF 8 or later.

RENAME-EXTERNAL

RENAME-EXTERN *objectfile old new* [*start*] [*end*]

Use RENAME-EXTERN to rename all occurrences of an external symbol from *old* to *new* in the specified modules.

Examples

The following option renames all external symbols in math.*rnn* from error to err:

```
RENAME-EXT math error err
```

Note: This option does not work with the output from modern IAR C/EC++ compilers or assemblers that produce UBROF 8 or later.

RENAME-GLOBAL

RENAME-GLOBAL *objectfile old new* [*start*] [*end*]

Use RENAME-GLOBAL to rename all occurrences of an external or public symbol from *old* to *new* in the specified modules.

Examples

The following option renames all occurrences of mean to average in math.*rnn*:

```
RENAME-GLOBAL math mean average
```

Note: This option does not work with the output from modern IAR C/EC++ compilers or assemblers that produce UBROF 8 or later.

RENAME-MODULE

RENAME-MODULE *objectfile old new*

Use RENAME-MODULE to rename a module. Notice that if there is more than one module with the name *old*, only the first one encountered is changed.

Examples

The following example renames the module average to mean in the file math.*rnn*:

```
RENAME-MOD math average mean
```

Note: This option does not work with the output from modern IAR C/EC++ compilers or assemblers that produce UBROF 8 or later.

RENAME-SEGMENT

RENAME-SEGMENT *objectfile old new* [*start*] [*end*]

Use RENAME-SEGMENT to rename all occurrences of a segment from the name *old* to *new* in the specified modules.

Examples

The following example renames all `CODE` segments to `ROM` in the file `math.rnn`:

```
RENAME-SEG math CODE ROM
```

Note: This option does not work with the output from modern IAR C/EC++ compilers or assemblers that produce UBROF 8 or later.

XLIB diagnostics

This chapter lists the messages produced by the IAR XLIB Librarian™.

XLIB messages

The following section lists the XLIB messages. Options flagged as erroneous never alter object files.

0 **Bad object file, EOF encountered**
Bad or empty object file, which could be the result of an aborted assembly or compilation.

1 **Unexpected EOF in batch file**
The last command in a command file must be `EXIT`.

2 **Unable to open file** *`file`*
Could not open the command file or, if `ON-ERROR-EXIT` has been specified, this message is issued on any failure to open a file.

3 **Variable length record out of bounds**
Bad object module, could be the result of an aborted assembly.

4 **Missing or non-default parameter**
A parameter was missing in the direct mode.

5 **No such CPU**
A list with the possible choices is displayed when this error is found.

6 **CPU undefined**
`DEFINE-CPU` must be issued before object file operations can begin. A list with the possible choices is displayed when this error is found.

7 **Ambiguous CPU type**
A list with the possible choices is displayed when this error is found.

8 **No such command**
Use the `HELP` option.

9 **Ambiguous command**
Use the `HELP` option.

10 **Invalid parameter(s)**
Too many parameters or a misspelled parameter.

11 **Module out of sequence**
Bad object module, could be the result of an aborted assembly.

12 **Incompatible object, consult distributor!**
Bad object module, could be the result of an aborted assembly, or that the assembler/compiler revision used is incompatible with the version of XLIB used.

13 **Unknown tag: hh**
Bad object module, could be the result of an aborted assembly.

14 **Too many errors**
More than 32 errors will make XLIB abort.

15 **Assembly/compilation error?**
The `T_ERROR` tag was found. Edit and re-assemble/re-compile your program.

16 **Bad CRC, hhhh expected**
Bad object module; could be the result of an aborted assembly.

17 **Can't find module: xxxxx**
Check the available modules with `LIST-MOD` file.

18 **Module expression out of range**
Module expression is less than one or greater than `$`.

19 **Bad syntax in module expression: xxxxx**
The syntax is invalid.

20 **Illegal insert sequence**
The specified destination in the `INSERT-MODULES` option must not be within the *`start-end`* sequence.

21 **<End module> found before <Start module>!**
Source module range must be from low to high order.

22 **Before or after!**
Bad `BEFORE/AFTER` specifier in the `INSERT-MODULES` option.

23 **Corrupt file, error occurred in** *`tag`*
A fault is detected in the object file *`tag`*. Reassembly or recompilation may help. Otherwise contact your supplier.

24 *`File`* **is write protected**
The file *`file`* is write protected and cannot be written to.

25 **Non-matching replacement module name found in source file**
In the source file, a module *`name`* with no corresponding entry in the destination file was found.

A

B

C

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Symbols