IAR Linker and Library Tools

Reference Guide

XLINK-5001

COPYRIGHT NOTICE

Copyright © 1987–2010 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems AB. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, From Idea To Target, IAR KickStart Kit, IAR PowerPac, IAR YellowSuite, IAR Advanced Development Kit, IAR, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB. J-Link is a trademark licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

All other product names are trademarks or registered trademarks of their respective owners.

EDITION NOTICE

April 2010

Part number: XLINK-5001

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

Contents

Tables vii

Preface ix

Who should read this guide ix
How to use this guide ix
What this guide contains x
Document conventions x
Typographic conventions xi
Naming conventions xi

Part 1: The IAR XLINK Linker 1

Introduction to the IAR XLINK Linker 3

Key features 3
Large Address Awareness 4
Linking protected files 4
MISRA C 4
The linking process 4
Object format 5
XLINK functions 5
Libraries 6
Output format 6
Input files and modules 6
Libraries 7
Segments 8
Segment control 9
Address translation 10
Allocation segment types 10
Memory segment types 11
Overlap errors 12
Range errors 12
Segment placement examples 13

Listing format 14
Header 14
Cross-reference 15
Checksummed areas and memory usage 22
Checksum calculation 22
Checksum calculation by the linker 23
Adding a checksum function to your source code 23
Things to remember 25
Checksum value symbol 26
Bytewise and mirrored initial checksum values 26
Bitwise initial values 26
Bytewise initial values 27
Mirroring 28
XLINK options 29
Setting XLINK options 29
Summary of options 29
Descriptions of XLINK options 31
Specifying the alignment of a segment 59
XLINK output formats 63
Single output file 63
UBROF versions 65
Two output files 66
Output format variants 67
IEEE695 68
ELF 70
XCOFF78K 72
Restricting the output to a single address space 73
XLINK environment variables 75
Summary of XLINK environment variables 75

XLINK diagnostics 79

Introduction 79

XLINK warning messages 79

XLINK error messages 79

XLINK fatal error messages 79

XLINK internal error messages 79

Error messages 80

Warning messages 97

Part 2: The IAR Library Tools 107

Introduction to the IAR Systems library tools 109

Libraries 109

IAR XAR Library Builder and IAR XLIB Librarian 109

Choosing which tool to use 110

Using libraries with C/C++ programs 110

Using libraries with assembler programs 110

The IAR XAR Library Builder 113

Using XAR 113

Basic syntax 113

Summary of XAR options 113

Descriptions of XAR options 114

XAR diagnostics 115

XAR messages 115

IAR XLIB Librarian options 117

Using XLIB options 117

Giving XLIB options from the command line 117

XLIB batch files 117

Parameters 118

Module expressions 118

List format 119

Using environment variables 119

Summary of XLIB options for all UBROF versions 120
Descriptions of XLIB options for all UBROF versions 121
Summary of XLIB options for older UBROF versions 130
Descriptions of XLIB options for older UBROF versions ... 131

XLIB diagnostics 133

XLIB messages 133

Index 135

Tables

1: Typographic conventions used in this guide xi
2: Naming conventions used in this guide xi
3: Allocation segment types 10
4: Memory segment types 11
5: Segment map (-xs) XLINK option 17
6: XLINK options summary 29
7: Disabling static overlay options 31
8: Disabling static overlay function lists 32
9: Arguments to --image_input 38
10: Checksumming algorithms 39
11: Checksumming flags 39
12: Mapping logical to physical addresses (example) 43
13: Enabling MISRA C rules (--misrac) 44
14: Disable range check options 51
15: Diagnostic control conditions (-ws) 55
16: Changing diagnostic message severity 55
17: Cross-reference options 56
18: XLINK formats generating a single output file 63
19: Possible information loss with UBROF version mismatch 66
20: XLINK formats generating two output files 66
21: XLINK output format variants 67
22: IEEE695 format modifier flags 68
23: IEEE695 format variant modifiers for specific debuggers 69
24: ELF format modifier flags 70
25: ELF format variant modifiers for specific debuggers 71
26: XCOFF78K format modifiers 72
27: XCOFF78K format variant modifiers for specific debuggers 72
28: XLINK environment variables 75
29: XAR parameters 113
30: XAR options summary 113
31: XLIB parameters 118

32: XLIB module expressions 118
33: XLIB list option symbols 119
34: XLIB environment variables 119
35: XLIB options summary 120
36: Summary of XLIB options for older compilers 130

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 Systems 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 5.0.0.1, 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 Systems 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 Systems linker and library tools, you should read the *Introduction to the IAR XLINK Linker* and *Introduction to the IAR Systems 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 Systems toolkit, we recommend that you first read the initial chapters of the *IAR Embedded Workbench® IDE 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® IDE User Guide* also contains complete reference information about the IAR Embedded Workbench IDE 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 Systems library tools* describes the IAR Systems library tools—IAR XAR Library Builder and IAR XLIB Librarian—which are designed to allow you to create and maintain relocatable libraries of routines.
- *The IAR XAR Library Builder* 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.
- *IAR XLIB Librarian 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

When, in this text, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `xxxxx\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 5.n\xxxxx\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file. Note that this style is also used for *`xxxxx`*, *`configfile`*, *`libraryfile`*, and other labels representing your product, as well as for the numeric part of filename extensions.
`[option]`	An optional part of a command.
`a|b|c`	Alternatives in a command.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

Brand name	Generic term
IAR Embedded Workbench® IDE	the IDE
IAR C-SPY® Debugger	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator

Table 2: Naming conventions used in this guide

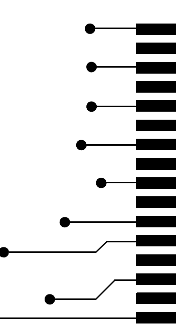
Brand name	Generic term
IAR C/C++ Compiler™	the compiler
IAR Assembler™	the assembler
IAR XLINK™ Linker	XLINK, the linker
IAR XAR Library builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

Table 2: Naming 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 computer using the IAR C-SPY Debugger Simulator.

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.

LARGE ADDRESS AWARENESS

XLINK is *Large Address Aware*, which means that XLINK can address 3 Gbytes of memory instead of the normal 2 if the host computer is prepared for this. Large Address Awareness is only relevant when linking very large projects where the memory requirements can exceed 2 Gbytes. Refer to Microsoft (*Memory Support and Windows Operating Systems*) for more details about this.

LINKING PROTECTED FILES

XLINK can link files protected by the License Management System. A protected file can only be successfully linked if a valid license can be found for that file. In the linker, it is only the IAR PowerPac™ object files that require a valid license. If one or more of the protected files contains a license requirement that cannot be satisfied, XLINK will generate error 160. License management in the linker is active only for protected files.

MISRA C

XLINK supports both MISRA C:2004 and MISRA C:1998. However, MISRA C:2004 does not introduce any new rules, only the rule numbers have been changed since MISRA C:1998.

MISRA C is a subset of C, suited for use when developing safety-critical systems, supported by some versions of IAR Embedded Workbench. The rules that make up MISRA C were published in "Guidelines for the Use of the C Language in Vehicle Based Software", and are meant to enforce measures for stricter safety in the ISO standard for the C programming language [ISO/IEC 9899:1990].

If your version of IAR Embedded Workbench supports checking for adherence to the MISRA C rules, you can set up the linker checking using the options `--misrac` and `--misrac_verbose`, see *--misrac*, page 44 and *--misrac_verbose*, page 45.

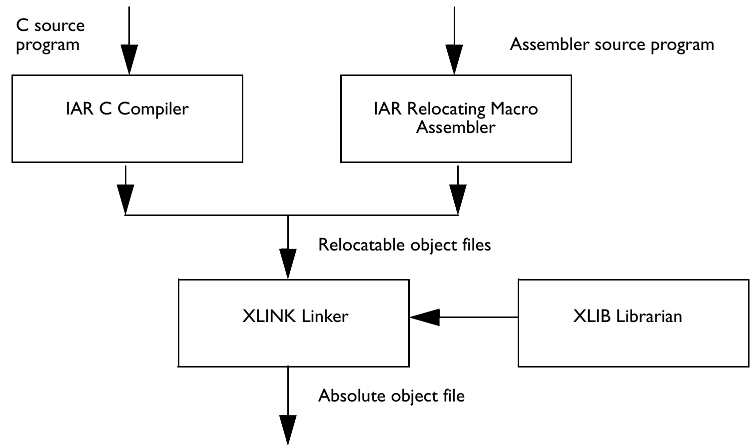
The implementation of the MISRA C rules does not affect code generation, and has no significant effect on the performance of IAR Embedded Workbench. The rules apply to the source code of the applications that you write and not to the code generated by the compiler. The compiler and linker only generate error messages, they do not actually prevent you from breaking the rules you are checking for.

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/C++, or mixed C/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/C++ programs.

XLINK FUNCTIONS

The IAR XLINK Linker performs four 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/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.

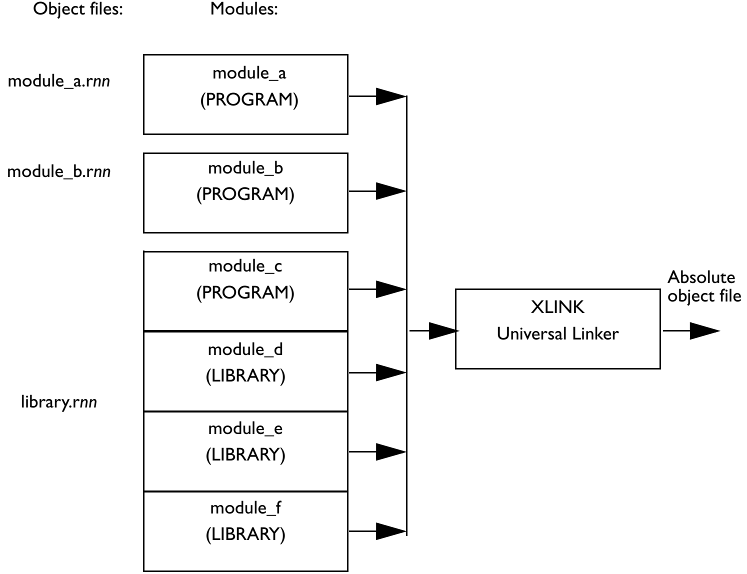
OUTPUT FORMAT

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 your PC using the IAR C-SPY Debugger Simulator.

Note: The default output format in 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/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/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/C++ source files and C/C++ modules. By default, the compiler gives this module the same name as the original C/C++ source file. Libraries of multiple C/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/C++ programs) has the `PROGRAM` attribute so that it will always get included when you link a C/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/C++ or assembler modules. The compiler option `--library_module` (`-b` for some IAR Systems products) can be used for making a C/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 assembler language programs the programmer is responsible for declaring and naming relocatable segments and determining how they are used. In C/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.

`-Ksegs=inc,count`	Duplicate code.
`-Ppack_def`	Define packed segments.
`-Zseg_def`	Define segments.
`-bbank_def`	Define banked segments.
`-Mrange_def`	Map logical addresses to physical addresses.

For detailed information about the options, see the chapter *XLINK options*, page 29.

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 47 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 3: 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 4: 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, see the description of *-z*, page 61.

RANGE ERRORS

Some instructions do not work unless a certain condition holds after linking, for example, that a branch target must be within a certain distance or that an address must be even. The compiler or assembler generates tests and XLINK verifies that the conditions hold when the files are linked. If a condition is not satisfied, XLINK generates a range error or warning and prints a description of the error.

Example

```
Error[e18]: Range error, chip’s branch target is out of range
  Where $ = vectorSubtraction + 0xC  [0x804C]
            in module "vectorRoutines" (vectorRoutines.r99),
            offset 0xC in segment part 5, segment NEARFUNC_A
  What: vectorNormalization - ($ + 8) [0x866B3FC]
  Allowed range: 0xFDFFFFFC - 0x2000000
  Operand: vectorNormalization [0x8673450]
           in module VectorNorm (vectorNormalization.r99),
           Offset 0x0 in segment part 0, segment NEARFUNC_V
```

Error[e18]: Range error

The first section is often the most important. The text after `Range error` is generated by the compiler and describes of what is being tested. In this case XLINK tests if the target of a branch instruction is in range.

Where

This is the location of the instruction that caused the range error. `$`, the address of the instruction, is `0x804c`, or `0xC` bytes after the label `vectorSubtraction`.

The instruction is in the module `vectorRoutines` in the object file `vectorRoutines.r99`. Another way to express the address where the instruction is located is as `0xC` bytes into segment part 5 of segment `NEARFUNC_A` of the `vectorRoutines` module. This can be helpful in locating the instruction in the rare cases when no label can be supplied.

What

This is the symbolic expression that XLINK evaluated and the value it resulted in. In this case, XLINK performs the calculation `0x8673450 - (0x804C + 8)` and gets the result `0x866B3FC`.

Allowed range

This is the range that the computed value was permitted to fall within. If the left hand side of the expression is greater than the right hand side, it should be interpreted as a negative value. In this case the range is `-0x2000004–0x2000000` and represents the reach of the processor's branch and link instruction.

Operand

Each symbolic operand in the expression is described in detail here. The format used is the same as in the definition of `$`.

Possible solutions

In this case the distance from the instruction in `vectorSubtraction` to `vectorNormalization` is too large for the branch instruction.

Possible solutions include placing the `NEARFUNC_V` segment closer to the segment `NEARFUNC_A` or using some other calling mechanism that can reach the required distance. It is also possible that the referring function tried to refer to the wrong target and that this caused the range error.

Different range errors have different solutions. Usually the solution is a variant of the ones presented above, in other words modifying either the code or the segment placement mechanism.

Note: Range error messages are not issued for references to segments of all types. See -*R*, page 51, for more information.

SEGMENT PLACEMENT 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
```

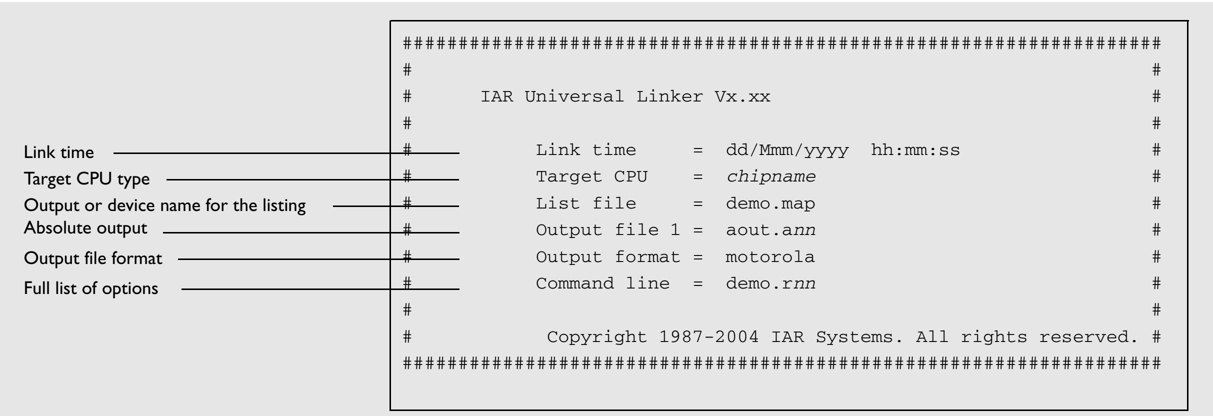
Segments SEGA and SEGB will be dumped at addresses 19000 to 1FFFF but the default 16-bit addressing mode will be used for accessing the data (i.e. 9000 to FFFF).

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 1987-2004 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.

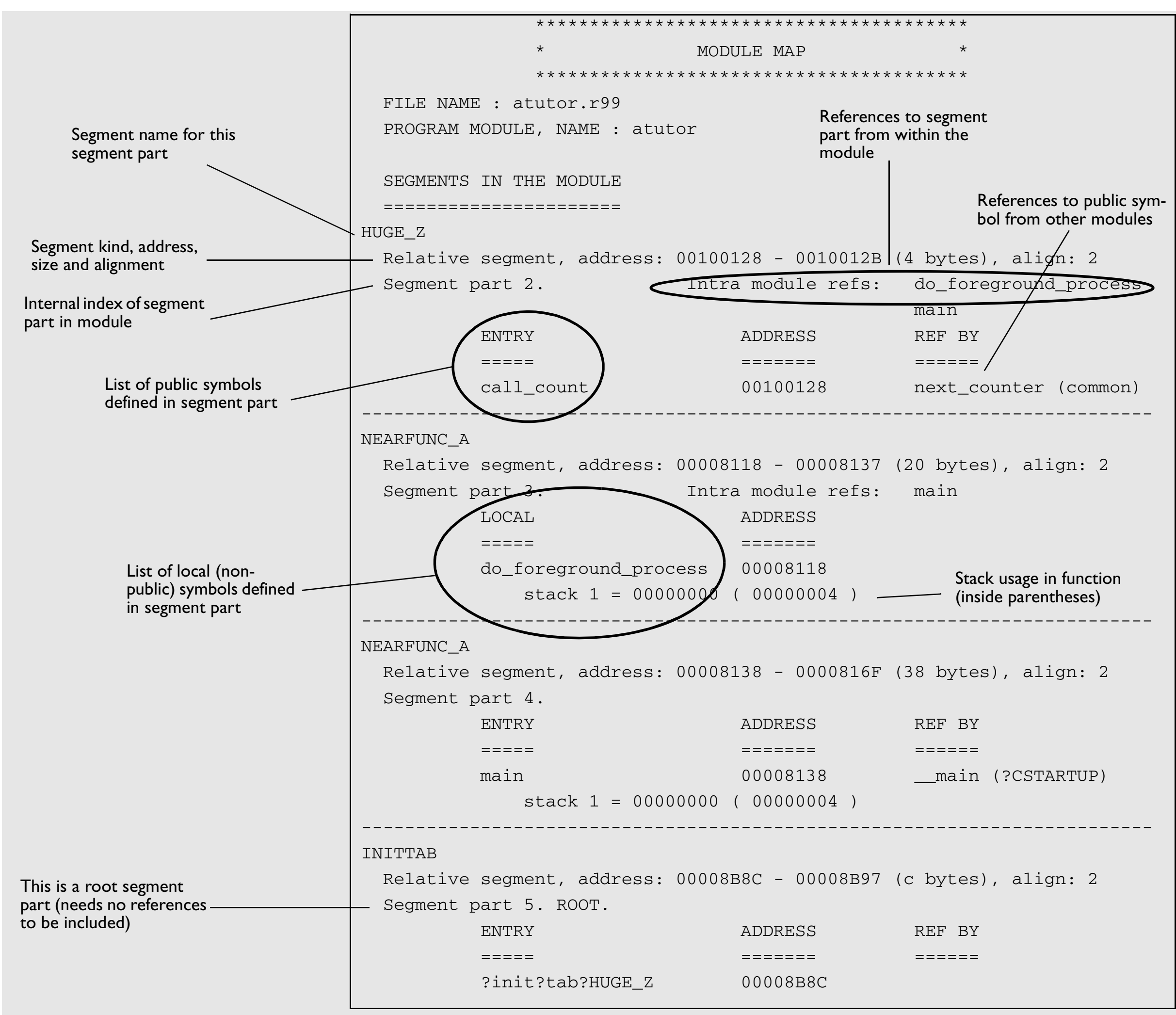
CROSS-REFERENCE

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 contains a list of files. For each file, those modules that were needed are listed. For each module, those segment parts that were included are listed. To also list the segment parts that were not included, use the `-xi` option. See *-x*, page 56.

The module map also contains a full cross reference, indicating for each segment part or symbol all references to it.

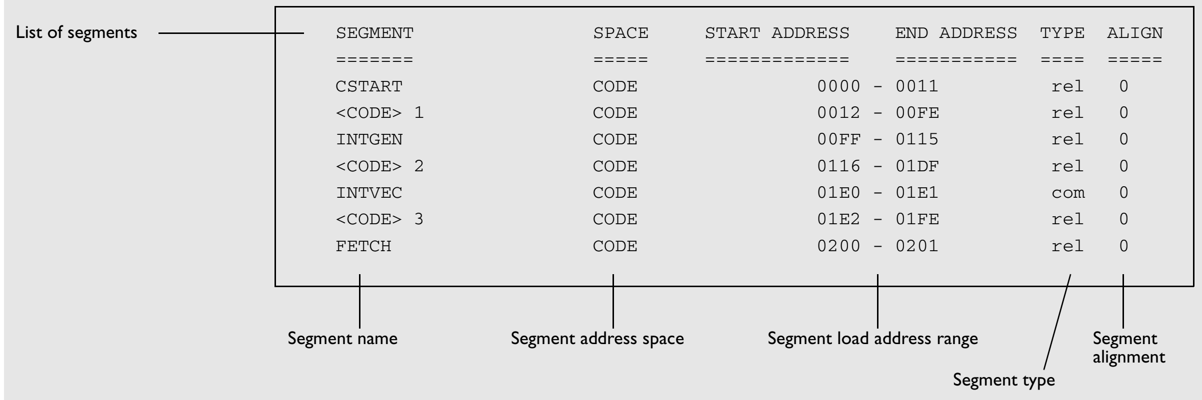


```
                 ****************************************
                 *              MODULE MAP              *
                 ****************************************

  FILE NAME : atutor.r99
  PROGRAM MODULE, NAME : atutor

  SEGMENTS IN THE MODULE
  ======================
HUGE_Z
  Relative segment, address: 00100128 - 0010012B (4 bytes), align: 2
  Segment part 2.          Intra module refs:   do_foreground_process
                                                main
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           call_count              00100128        next_counter (common)
-------------------------------------------------------------------------
NEARFUNC_A
  Relative segment, address: 00008118 - 00008137 (20 bytes), align: 2
  Segment part 3.          Intra module refs:   main
           LOCAL                   ADDRESS
           =====                   =======
           do_foreground_process   00008118
               stack 1 = 00000000 ( 00000004 )
-------------------------------------------------------------------------
NEARFUNC_A
  Relative segment, address: 00008138 - 0000816F (38 bytes), align: 2
  Segment part 4.
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           main                    00008138        __main (?CSTARTUP)
               stack 1 = 00000000 ( 00000004 )
-------------------------------------------------------------------------
INITTAB
  Relative segment, address: 00008B8C - 00008B97 (c bytes), align: 2
  Segment part 5. ROOT.
           ENTRY                   ADDRESS         REF BY
           =====                   =======         ======
           ?init?tab?HUGE_Z        00008B8C
```


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:



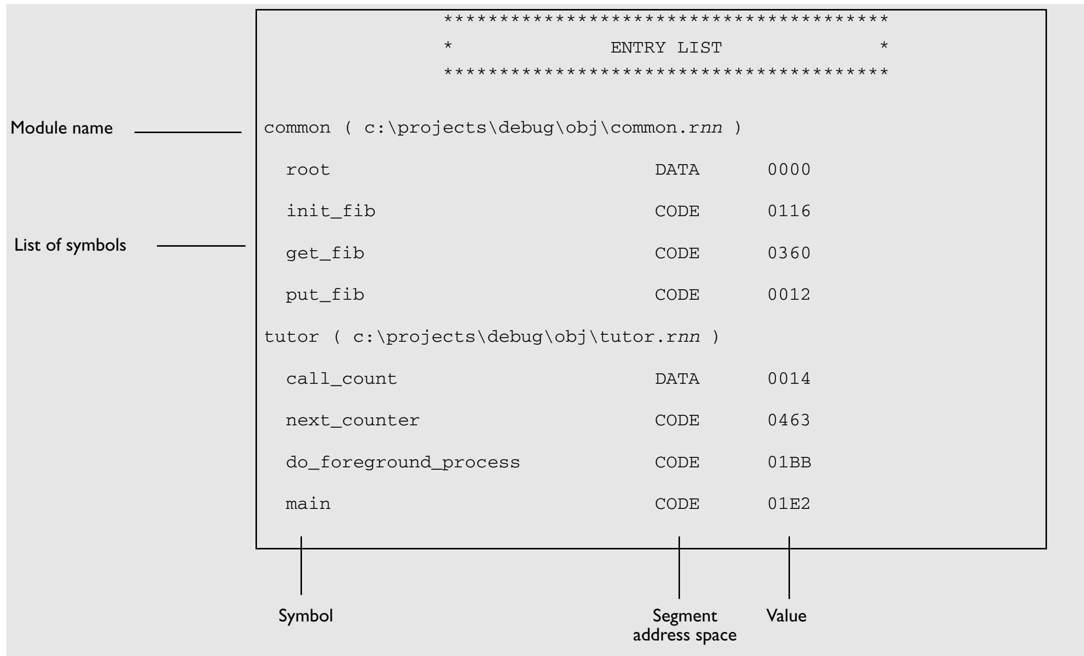
This lists the following:

Parameter	Description
SEGMENT	The segment name.
SPACE	The segment address space, usually CODE or DATA.
START ADDRESS	The start of the segment's load address range.
END ADDRESS	The end of the segment's load address range.
TYPE	The type of segment: rel Relative stc Stack. bnk Banked. com Common. dse Defined but not used.
ALIGN	The segment is aligned to the next 2^ALIGN address boundary.

Table 5: Segment map (-xs) XLINK option

Symbol listing (-xe)

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



Module summary (-xn)

The *module summary* summarizes the contributions to the total memory use from each module. Each segment type that is used gets a separate column, with one or two sub-columns for relocatable (`Rel`) and absolute (`Abs`) contributions to memory use.

Only modules with a non-zero contribution to memory use are listed. Contributions from `COMMON` segments in a module are listed on a separate line, with the title `+ common`.

Contributions for segment parts defined in more than one module and used in more than one module are listed for the module whose definition was chosen, with the title `+ shared`:

```
                ****************************************
                *            MODULE SUMMARY            *
                ****************************************

Module            CODE   DATA  CONST
------            ----   ----  -----
                 (Rel)  (Rel)  (Rel)
?CSTARTUP          152
?Fclose            308
?Fflush            228
?Fputc             156
?Free              252
?INITTAB                           8
?Malloc            348      8
?Memcpy             36
?Memset             28
?Putchar            28
?RESET
  + common           4
?Xfiles                   376    296
  + shared                        12
?Xfwprep           284
?Xgetmemchunk       96      1
?_EXIT              72
?__dbg_Break         4
?__exit             28
?close              36
?cppinit           100      4
?d__write           44
?div_module        100
?exit               20
?heap                              8
?low_level_init      8
?remove             36
?segment_init      120
```

```
?write              20
atutor              88      4
  + shared                         12
atutor2            364     40
                 -----    ---     ---
Total:           2 960    433     336
```

Static overlay system map (-xo)

If the **Static overlay system map** (`-xo`) option has been specified, the list file includes a listing of the static overlay system.

Static overlay is a system used by some IAR Systems compilers, where local data and function parameters are stored at static locations in memory. The linker's static overlay process lays out the overlay areas—memory areas for parameters and local data—for each function so they do not overlap the overlay areas for other functions that might be called at the same time.

The listing is separated into one section for each sub-tree of the function call tree. At the top of each section, the stack segment and overlay segment used are listed.

Each sub-tree section shows either the functions that can be reached from a root function or the functions that can be reached from a function that can be called indirectly. Called functions are listed before the calling function, and relationships are displayed using indentation and numbering.

For each function, information is listed first about stack usage and then about the overlay area. The stack usage information includes previous stack usage and how much stack the current function block uses. The static overlay information includes the start location of the area where parameters and local data are placed, and the amount of memory used in the current function. The most important information is the static overlay address; it is used by your application and must be correct.

Example of a sub-tree section:

```
->Sub-tree of type: Function tree
                         CALLSTACK
  | Stack used (prev) :  00000000
                         <OVERLAY0,WRKSEG> 1
  | Stat overlay addr :  00000066
03      func_1
        | Stack used (prev) :  00000000
        | + function block  :  00000002
        | Stat overlay addr :  00000066
        | + in function     :  00000002
03      func_2
        | Stack used (prev) :  00000000
        | + function block  :  00000002
```

```
        | Stat overlay addr :  00000066
        | + in function     :  00000001
        |  Already listed
02    main
      | Stack used (prev) :  00000002
      | + function block  :  00000004
      | Stat overlay addr :  00000068
      | + in function     :  00000006
01  __CSTARTUP
    | Stack used (prev) :  00000006
    | + function block  :  00000000
    | Stat overlay addr :  0000006E
    | + in function     :  00000000
<-Sub-tree of type: Function tree
  | Stack used          :  00000006
  | Static overlay acc. :  0000006E
```

In this example, `main` calls the functions `func_1` and `func_2`. `__CSTARTUP` is the root of this function call sub-tree and is a function in the runtime library which calls the `main` function of your application.

`func_1` needs 2 bytes of stack in the stack segment—`CALLSTACK`—and a 2-byte overlay area in the overlay segment `<OVERLAY0,WRKSEG> 1` (the result of packed placement of `OVERLAY0` and `WRKSEG`). The parameters and local variables (2 bytes) of `func_1` are placed at address `0x66`.

`func_2` also needs 2 bytes of stack, but a 1-byte overlay area. The parameters and local variables of `func_2` are also placed at address `0x66`, as `func_1` and `func_2` are independent of each other.

`main` needs 4 bytes of stack and a 6-byte overlay area. Because the overlay area of the `main` function must not overlap the overlay area of either `func_1` or `func_2`, it is placed at address `0x68`.

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.

Checksum calculation

XLINK can be set up to generate a checksum that can be compared to a checksum calculated by your application or any other checksum calculating process that can checksum the generated image.

To use checksumming to verify the integrity of your application, you must:

- Set up XLINK to generate a checksum and make sure the checksum bytes are included in the application by placing the checksum in a named segment and giving it a name, for details see the XLINK option *-J*, page 38. See also the option *-Z*, page 58 for information about placing the checksum in a segment.
- Choose a checksum algorithm and include source code for the algorithm in your application.
- Decide what memory ranges to verify and set up the source code for it in your application source code.

CHECKSUM CALCULATION BY THE LINKER

Checksum calculation in the linker can be set up in the IDE or by using the `-J` option. By default the calculated checksum is placed in the segment `CHECKSUM`, and the symbol `__checksum` is defined.

To set up calculation of the checksum in the IDE, choose **Project>Options>Linker>Processing**

Example 1

For example, to calculate a 2-byte checksum using the generating polynomial `0x11021` and output the one's complement of the calculated value, specify:

```
-J2,crc16,1
```

All available bytes in the application are included in the calculation.

Example 2

```
-J2,crc16,2m,lowsum=(CODE)0-FF
```

This example calculates a checksum as above, located in a 2-byte segment part in the `CHECKSUM` segment, with the following differences: The output is the mirrored 2's complement of the calculation. The symbol `lowsum` is defined and only bytes in the range `0x0-FF` in the `CODE` address space are included.

Example 3

```
-J2,crc16,,highsum,CHECKSUM2,2=(CODE)F000-FFFF;(DATA)FF00-FFFF
```

This example calculates a checksum as above, now based on all bytes that fall in either of the ranges given. It is placed in a 2-byte segment part with an alignment of 2 in the segment `CHECKSUM2`, and the symbol `highsum` is defined.

ADDING A CHECKSUM FUNCTION TO YOUR SOURCE CODE

To check the value of the checksum generated by XLINK, the checksum must be compared with a checksum that your application has calculated. This means that you must add a function for checksum calculation (that uses the same algorithm as the checksum generated by XLINK) to your application source code, or use some kind of hardware CRC. Your application must also include a call to this function.

A function for checksum calculation

This function—a slow variant but with small memory footprint—uses the CRC16 algorithm:

```
unsigned short slow_crc16(unsinged short sum, unsigned char *p,
                          unsigned int len)
{
  while (len--)
    {
      int i;
      unsigned char byte = *(p++);
      for (i = 0; i < 8; ++i)
        {
          unsigned long osum = sum;
          sum <<= 1;
          if (byte & 0x80)
            sum |= 1;
          if (osum & 0x8000)
            sum ^= POLY;
          byte <<= 1;
        }
     }
   return sum;
}
```

`POLY` is the generating polynomial. The checksum is the result of the final call to this routine.

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 `slow_crc16` function 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 (or the specified initial value).

Calculating a checksum in your source code

This source code gives an example of how the checksum can be calculated:

```
/* Start and end of the checksum range */
/* Must exclude the checksum itself */
unsigned long ChecksumStart = 0x8000+2;
unsigned long ChecksumEnd = 0x8FFF;

/* The checksum calculated by XLINK */
extern unsigned short __checksum;

void TestChecksum()
{
  unsigned short calc = 0;

  /* Run the checksum algorithm */
  calc = slow_crc16(0,
                    (unsigned char *) ChecksumStart,
                     (ChecksumEnd - ChecksumStart+1));

  /* Rotate out the answer */
  unsigned char zeros[2] = {0, 0};
  calc = slow_crc16(calc, zeros, 2);

  /* Test the checksum */
  if (calc != __checksum)
  {
    abort();   /* Failure */
  }
}
```

THINGS TO REMEMBER

When calculating a checksum, you must remember that:

- The checksum must be calculated from the lowest to the highest address for every memory range
- Each memory range must be verified in exactly the same order as defined
- It is OK to have several ranges for one checksum
- If several checksums are used, you should place them in sections with unique names and use unique symbol names
- If the slow CRC function is used, you must make a final call to the checksum calculation with as many bytes (with the value `0x00`) as you have bytes in the checksum.

CHECKSUM VALUE SYMBOL

If you want to verify that the contents of the target ROM and the debug file are the same, use the *checksum value symbol*, `__checksum__value`. A generated output file in UBROF or ELF/DWARF format contains a checksum value symbol for each checksum symbol (see *sym*, page 39). The checksum value symbol helps the debugger to see if the code in target ROM corresponds to the code in the debug file. Because this symbol is added after linking, it cannot be accessed from your application, and its only use is to verify that the ROM content in a file is identical to that of the debug file.

The checksum value symbol has the same name as the checksum symbol, with `__value` added at the end. For example, for the default checksum symbol `__checksum`, the checksum value symbol will be `__checksum__value`.

The value of `__checksum__value` is the checksum generated by the checksum option `-J`. It is not the address of the checksum bytes, but the value of the checksum symbol.

If the CRC16 checksum for a certain application is `0x4711`, located at address `0x7FFE`, the output file will, by default, contain the symbol `__checksum` with the value `0x7FFE` and the symbol `__checksum__value` with the value `0x4711`.

Note: In some cases, the code can be different even when the values of the checksum value symbol are identical. One such case is when position-independent code is located at different addresses in different output images, as the checksum only depends on the contents of bytes and not on their addresses.

Bytewise and mirrored initial checksum values

It is possible to specify bytewise initial values and mirrored initial values. Every bytewise and mirrored initial value can be expressed equally well as a bitwise non-mirrored initial value. Specifying bytewise and mirrored initial values is simply a convenient way to specify the same initial value both in XLINK and in the verification step in the application or image loader, in cases where the verification step uses bytewise or mirrored initial values. The application can checksum itself, or an image loader can checksum the application.

Mirroring is the process of reversing all the bits in a binary number, see *Mirroring*, page 28.

BITWISE INITIAL VALUES

If a bitwise initial value is specified in the checksum command, that value is used as the initial value of `sum`, see the classic bit-by-bit calculation in *A function for checksum calculation*, page 24.

For an `n`-byte checksum you need to feed `n * 8` zero bits through the bit-by-bit algorithm after the last bit has been entered. This allows the last `n * 8` bits of the checksum to be rotated out of the checksum algorithm.

Example

This example specifies a 2-byte CRC16 checksum where the initial value of `sum` in the previous bit-by-bit C function is `0x4711`.

```
-J2,crc16,,,,,0x4711
```

Note: The bit-by-bit algorithm is also called *slow CRC*. Bitwise initial values are sometimes called *indirect initial values* in texts about CRC.

BYTEWISE INITIAL VALUES

If a bytewise initial value is specified on the command line, that value is used as the initial value of `sum` in this byte-by-byte calculation:

```
unsigned short
byte_by_byte_crc(uint16_t sum, uint8_t *p, unsigned int len)
{
  while (len--)
    sum = table[sum >> 8) ^ *p++] ^ (sum << 8);
  return sum;
}
```

Note: The byte-by-byte algorithm does not need any final zero bits.

Byte-by-byte CRC algorithms execute faster than bit-by-bit CRC algorithms, but use more space. They use a table of precomputed CRC values. For more information about CRC tables, see the examples in *Technical Note 91733* available on the IAR Systems web site.

Example

This example specifies a 2-byte CRC16 checksum where the initial value of `sum` in the byte-by-byte C function is `0x1D0F`:

```
-J2,crc16,,,,,#0x1D0F
```

The byte-by-byte algorithm computes exactly the same checksum as the bit-by-bit algorithm (once the final zeroes have been fed through the bit-by-bit algorithm). They cannot use the same initial value due to differences in how the initial values are handled.

Note: The byte-by-byte algorithm is called *fast CRC*. Bytewise initial values are sometimes called *direct initial values* in texts about CRC.

MIRRORING

Mirroring is the process of reversing all the bits in a binary number. If the number has *n* bits, bit 0 and bit *n*-1 are swapped, as are bits 1 and *n*-2 and so on.

To specify a mirrored initial value, use the `m` prefix, see the option -*J*, page 38.

Example 1

```
mirror(0x8000) = 0x0001
mirror(0xF010) = 0x080F
mirror(0x00000002) = 0x40000000
mirror(0x12345678) = 0x1E6A2C48
```

Example 2

This example specifies a 2-byte CRC checksum with the bytewise initial value `0x5012` (`0x480A` interpreted as a 16-bit binary number and mirrored):

```
-J2,crc16,,,,,m0x480A
```

In XLINK, the size of the checksum determines the number of bits in the initial value that will be mirrored. `-J4,...,m0x2000` specifies the bitwise initial value `0x00040000`, not `0x0004`, because the initial value is treated as a 4-byte quantity when the size of the checksum is 4 bytes.

Note: Mirroring is sometimes called *reflection* in texts about CRC.

XLINK options

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

The *IAR Embedded Workbench® IDE User Guide* describes how to set XLINK options in the IAR Embedded Workbench IDE, 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
`-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 6: XLINK options summary

Command line option	Description
-F*format*	Specifies output format
-f *file*	Specifies XCL filename
-G	Disables global type checking
-g*symbol1*[,*symbol2*,*symbol3*,...]	Requires global entries
-H*hexstring*	Fills unused code memory
-h*[(seg_type)]{range}*	Fills ranges.
-I*pathname*	Includes paths
--image_input	Links pure binary files
-J*size*,*algo*[,*flags*][=*ranges*]	Generates a 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
--misrac	Enables MISRA C-specific error messages
--misrac_verbose	Enables verbose logging of MISRA C checking
-n[c]	Ignores local symbols
-O*format*[,*variant*][=*filename*]	Multiple output files
-o *file*	Output file
-P*pack_def*	Defines packed segments
-p*lines*	Specifies lines/page
-Q	Scatter loading
-q	Disables relay function optimization
-R[w]	Disables range check
-r	Debug information
-rt	Debug information with terminal I/O
-S	Silent operation
-s *symbol*	Specifies new application entry point
-U[(*address_space*)]*range*=[(*address_space*)]*range*	Address space sharing
-V(*type*)*name*[,*align*]	Declares relocation areas for code and data
-w[n|s|t|ID[=*severity*]]	Sets diagnostics control
-x[e][h][i][m][n][s][o]	Specifies cross-reference

Table 6: XLINK options summary (Continued)

Command line option	Description
`-Y[char]`	Format variant
`-y[chars]`	Format variant
`-Z[@]seg_def`	Defines segments
`-z[b]`	Segment overlap warnings

Table 6: XLINK options summary (Continued)

Descriptions of XLINK options

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

-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.

-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 7: 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.

Table 8: Disabling static overlay function lists

Function list	Description
[*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 8: Disabling static overlay function lists (Continued)

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 linker category in the IAR Embedded Workbench IDE.

-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 49.

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 9.

Note: This option is included for backward compatibility reasons. We recommend that you instead use `-P` to define packed segments. See page 49.

-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.

-c `-cprocessor`

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 IDE.

-D `-Dsymbol=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 linker category in the IAR Embedded Workbench IDE.

-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
```

-e `-enew=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 `-Fformat`

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 linker category in the IAR Embedded Workbench IDE.

-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 identical to the **Linker command file** option in the linker category in the IAR Embedded Workbench IDE.

`-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 linker category in the IAR Embedded Workbench IDE.

`-g` `-g`*`symbol1`*`[,`*`symbol2`*`,`*`symbol3`*`,…]`

XLINK normally only includes segment parts (usually functions and variables) that are needed, to satisfy all references from segment parts that must be included. Use the `-g` option to add to this set so that something is included even if it appears not to be needed.

`-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 linker category in the IAR Embedded Workbench IDE.

`-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.

For 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 path name 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.

For example, to read object files from `v:\general\lib` and `c:\project\lib`:

```
-Iv:\general\lib
-Ic:\project\lib
```

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

This option is identical to the **Search paths** option in the linker category in the IAR Embedded Workbench IDE.

`--image_input` `--image_input=filename,symbol,segment,alignment`

Use this option to link pure binary files in addition to the ordinary input files.

Parameter	Description
filename	The pure binary file you want to link
segment	The segment where the binary data will be placed.
symbol	The symbol defined by the segment part where the binary data is placed
alignment	The alignment of the segment part where the binary data is placed

Table 9: Arguments to --image_input

The file's entire contents is placed in the segment, which means it can only contain pure binary data (for instance, the raw-binary output format, see *Single output file*, page 63).

The segment part where the contents of the *filename* file is placed, is only included if the symbol *symbol* is required by your application. Use the `-g` option if you want to force a reference to the segment part, see *-g*, page 36.

Example

```
--image_input=bootstrap.rnn,Bootstrap,CSTARTUPCODE,4
```

The contents of the pure binary file `bootstrap.rnn` is placed in the segment `CSTARTUPCODE`. The segment part where the contents is placed will be 4-byte aligned and will only be included if your application (or the command line option `-g`) includes a reference to the symbol `Bootstrap`.

This option corresponds to the **Raw binary image** option in the linker category in the IAR Embedded Workbench IDE.

`-J` `-Jsize,algo[,flag[,sym[,seg[,align[,[m][#]val]]]]][{=|==}ranges]`

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

size

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

algo

`algo` 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 10: Checksumming algorithms

flag

`flag` 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 11: Checksumming flags

sym

`sym` is an optional user-defined symbol name for the checksum. If you specify the symbol name yourself, the checksum it symbolizes is only included in the final application if it is referenced by any included parts of the application, or if the `-g` (require global entries) command line option is used for the symbol. If you do not specify a symbol explicitly, the name `__checksum` is used. In this case the symbol is always included in the final application.

seg

`seg` is an optional user-defined name for the segment to put the checksum in. If you do not specify a segment name explicitly, the name `CHECKSUM` is used. This segment must be placed using the segment placement options like any other segment.

align

`align` is an optional user-defined alignment for the checksum. If you do not specify an alignment explicitly, an alignment of 1 is used.

m

`m` specifies that the initial value *val* will be mirrored before it is used. `m0x2468` is a bitwise initial value that will be mirrored (`0x2468` before mirroring), and `m#0x8C18` is a bytewise initial value that will be mirrored (`0x8C18` before mirroring). See *Mirroring*, page 28.

#

`#` specifies that *val* is a bytewise initial value. `#0x1D0F` is an example of a bytewise initial value. Bytewise initial values are used if the verification step uses the byte-by-byte algorithm with non-zero initial values. See *Bytewise initial values*, page 27.

val

val is the initial value in hexadecimal form. By default, the initial value of the checksum is 0. If you need to change the initial value, supply a different value for *val*. If nothing else is specified, the initial value is a bitwise initial value. For example, `0x4711` is a bitwise initial value. See *Bitwise initial values*, page 26.

ranges

ranges is one of:

- one or more explicit ranges of hexadecimal addresses, like `200–5FF`
- one or more symbolic ranges, like `CHECKSUM_START–CHECKSUM_END`
- one or more segment names inside a `{}` pair, like `{CODE}`.

If more than one address range is specified, the ranges are comma-separated.

Note: If you do not specify any ranges explicitly, all bytes in the final application are included in the calculation.

=

If one equal sign is used (`=`), a range is only checksummed once, even if it is specified more than once. An example:

```
-J2,crc16=CRC_START-CRC_END,40-7F,{CODE}
```

If `CRC_START` has the value `40`, `CRC_END` has the value `4F`, and the segment `CODE` is placed in the address range `50–113`, this checksum command is equivalent to:

```
-J2,crc16=40-4F,40-7F,50-113
```

The ranges overlap, so this is equivalent to:

```
-J2,crc16=40-113
```

The checksumming will start on address `40` and end on address `113`. The byte on each address will be used exactly once, even though some address ranges were specified more than once.

==

If two equal signs are used (`=`), a range is checksummed as many times as it is specified and in the specified order. If, for instance, two equal signs are used in the example above:

```
-J2,crc16==CRC_START-CRC_END,40-7F,{CODE}
```

this checksum command is equivalent to:

```
-J2,crc16==40-4F,40-7F,50-113
```

Because two equal signs are used instead of a single equal sign, the bytes will be checksummed in this order:

1. The bytes in `40-4F` are checksummed
2. The bytes in `40-4F` are checksummed a second time (the first half of the range `40-7F`)
3. The bytes in `50-7F` are checksummed (the second half of the range `40-7F`)
4. The bytes in `50-7F` are checksummed a second time (the first half of the range `50-113`)
5. The bytes in `80-113` are checksummed (the second half of the range `50-113`).

To set up calculation of the checksum in the IDE, choose **Project>Options>Linker>Processing**

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

Use `-K` to duplicate any raw data bytes from the segments in *`segs`* *`count`* times, adding/subtracting *`diff`* to/from 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.

Example 1

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`.

Example 2

If the segment `MYSEG` is placed at `0x10000`, to create 4 duplicates placed at `0xE000`, `0xC000`, `0xA000`, and `0x8000`, specify:

```
-KMYSEG=-0x2000,4
```

For more information, see *Segment control*, page 9.

`-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 linker category in the IAR Embedded Workbench IDE.

`-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 linker category in the IAR Embedded Workbench IDE.

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

--misrac

`--misrac[={tag1,tag2-tag3,…|all|required}]`

If your version of IAR Embedded Workbench supports MISRA C, use this option to enable the linker to check for deviations from the rules described in the MISRA *Guidelines for the Use of the C Language in Vehicle Based Software*. By using one or more arguments with the option, you can restrict the checking to a specific subset of the MISRA C rules. The possible arguments are described in this table:

Command line option	Description
`--misrac`	Enables checking for all MISRA C rules
`--misrac=n`	Enables checking for the MISRA C rule with number *n*
`--misrac=m,n`	Enables checking for the MISRA C rules with numbers *m* and *n*
`--misrac=k-n`	Enables checking for all MISRA C rules with numbers from *k* to *n*
`--misrac=k,m,r-t`	Enables checking for MISRA C rules with numbers *k*, *m*, and from *r* to *t*
`--misrac=all`	Enables checking for all MISRA C rules
`--misrac=required`	Enables checking for all MISRA C rules categorized as required

Table 13: Enabling MISRA C rules (--misrac)

If the linker is unable to check for a rule, specifying the option for that rule has no effect. Because most rules are concerned with either compiling or documenting, only rules 11, 23, 25, and 26 can be checked by the linker. As a consequence, specifying, for example, `--misrac=15` has no effect.

When a rule is violated, XLINK generates an error. This can be changed by using the usual diagnostics control mechanism. See the chapter *XLINK diagnostics*.

This option is related to the **MISRA C** options in the General category in the IAR Embedded Workbench IDE.

--misrac_verbose

`--misrac_verbose`

If your version of IAR Embedded Workbench supports MISRA C, use this option to generate a MISRA C log during linking. This is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.

If this option is enabled, the linker displays a text at sign-on that shows both enabled and checked MISRA C rules.

This option is related to the **MISRA C** options in the General category in the IAR Embedded Workbench IDE.

-N

`N` *`filename`*`[,`*`filename`*`,`*`filename`*`,...]`

Use `-N` to specify that all content in one or more files is treated as if it had the `root` attribute. This means that it is included in the application whether or not it is referenced from the rest of the application.

Note: Modules will still be removed at link time if they are not referenced, so `root` content in a non-referenced module will not be included in the application. Use the linker options `-A myFile.r99` and `-N myFile.r99` at the same time to make sure that all modules in the file are kept and that all content in the file is treated as `root`.

-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 linker category in the IAR Embedded Workbench IDE.

-O `-Oformat[,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
-Oelf,as=..\MyElfCode\myApp.elf
```

This will result in:

- one output file named `foo.dbg`, using the UBROF format
- one output file named `foo.s99`, using the MOTOROLA format
- one output file named `abs.x`, using the INTEL-EXTENDED format just as if `-Y1` had also been specified
- one output file named `myApp.elf` created in the overlying directory `MyElfCode`, using the ELF format just as if `-yas` 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 IDE.

Note: If `-r` is specified—or its corresponding option in the IAR Embedded Workbench IDE—only one output file is generated, using the UBROF format and selecting special runtime library modules for IAR C-SPY.

This option is related to the **Extra output** options in the linker category in the IAR Embedded Workbench IDE.

-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 35, 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 linker category in the IAR Embedded Workbench IDE.

-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.
	start:+*size*	The range starting at *start* with the length *size*.

`[start:+size]*count+offset`	Specifies *count* ranges, where the first begins at *start* and has the length *size*, the next one begins at *start*+ *offset* and has the same length, and so on. The `+offset` part is optional, and defaults to the length of the range.
`[start:+size]/pagesize`	Specifies the entire range beginning at *start* and with the length *size*, divided into pages of size and alignment *pagesize*. *Note*: The beginning 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* (or *size*) 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`.
`1000:+1000`	One range from `1000` to `1FFF`.
`[1000:+1000]*3+2000`	Three ranges: `1000-1FFF`,`3000-3FFF`,`5000-5FFF`.

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. If you want, you can put `0x` before the number to make it extra clear that it is hexadecimal, like this: `0x4FFF`.

For more information see *Segment control*, page 9.

-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 identical to the **Lines/page** options in the linker category in the IAR Embedded Workbench IDE.

-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.

Example

This example is not intended as a guide to writing code that is copied from ROM to RAM, but as an example of how it *can* be done without using the assembler. All you need to add to the example is the -Q command and the placement commands for the segments RAMCODE and ROMCODE.

```
/* include memcpy */
#include <string.h>

/* declare that there exist 2 segments, RAMCODE and ROMCODE */
#pragma segment="RAMCODE"
#pragma segment="ROMCODE"

/* place the next function in RAMCODE */
#pragma location="RAMCODE"

/* this function is placed in RAMCODE, it does nothing useful,
   it's just an example of an function copied from ROM to RAM */

int adder(int a, int b)
{
  return a + b;
}

/* enable IAR extensions, this is necessary to get __sfb and
__sfe, it is of course possible to write this function in
assembler instead */
#pragma language=extended

void init_ram_code()
{
  void * ram_start   = __sfb("RAMCODE");  /* start of RAMCODE */
  void * ram_end     = __sfe("RAMCODE");  /* end of RAMCODE */
  void * rom_start   = __sfb("ROMCODE");  /* start of ROMCODE */

  /* compute the number of bytes to copy */
  unsigned long size = (unsigned long)(ram_end) - (unsigned
long)(ram_start);

  /* copy the contents of ROMCODE to RAMCODE */
  memcpy( ram_start, rom_start, size );
}
```

```
/* restore the previous mode */
#pragma language=default

int main()
{
  /* copy ROMCODE to RAMCODE, this needs to be done before
anything in RAMCODE is called or referred to */
  init_ram_code();

  /* call the function in RAMCODE */
  return adder( 4, 5 );
}
```

-q `-q`

When used with version 4.10 or later of the ARM IAR C/C++ Compiler, XLINK performs relay function optimization. Using the `-q` option disables this optimization, retaining all used relay functions in the program. The option has no effect if there are no relay functions to optimize.

-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 assembler 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 14: 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** option in the linker category in the IAR Embedded Workbench IDE.

-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 linker category in the IAR Embedded Workbench IDE.

-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 linker category in the IAR Embedded Workbench IDE.

-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.

-s `-s` *`symbol`*

This option adds a new way to specify the entry point for an application.

If the option is used, the specified *`symbol`* will be used as the application entry point, and there must be a definition for the symbol in the application, or an `Undefined External` error (error 46) will be generated. This symbol definition will also be included in the final link image.

This option is identical to the **Override default program entry** option in the linker category in the IAR Embedded Workbench IDE.

-U `-U[(address_space)]range=[(address_space)]range`

where the parameters are as follows:

address_space		Specifies the address space if applicable for the target processor. If omitted it defaults to `CODE`.
range	*start-end*	The range starting at *start* and ending at *end*.

Each `-U` command line option declares that the memory given by the range on the left side of the `=` is the same memory as that given by the range on the right side. This has the effect that, during segment placement, anything occupying some part of either memory will be considered to reserve the corresponding part of the other memory as well.

The optional segment type parameter `(address_space)` that can be included on each side of the `=` can be used to specify the address space for architectures with multiple address spaces.

Example

This example assumes an architecture with separate code and address spaces, where the `CODE` segment type corresponds to the code address space and the `DATA` segment type to the data address space.

```
-U(CODE)4000-5FFF=(DATA)11000-12FFF
-P(CODE)MYCODE=4000-5FFF
-P(DATA)MYCONST=11000-12FFF
```

The first line declares that the memory at `4000–5FFF` in the code address space is also mapped at `11000–12FFF` in the data address space.

The second line places the `MYCODE` segment at `4000–5FFF` in the code address space. The corresponding bytes in the data address space will also be reserved. If `MYCODE` occupies the addresses `4000–473F`, the range `11000–1173F` in the data address space will also be reserved.

The third line will place the `MYCONST` segment into whatever parts of the `11000–12FFF` memory range are not reserved. In this case it will behave as if we had written:

```
-P(DATA)MYCONST=11740-12FFF
```

`-U` is not transitive. This means that overlapping address spaces specified by the same placement option will not be distributed correctly to all involved address ranges. See this example:

```
-U(CODE)1000-1FFF=(DATA)20000-20FFF
```

```
-U(DATA)20000-20FFF=(CONST)30000-30FFF
```

In this example, if some bytes are placed in the CODE space at address 1000, the corresponding bytes in the DATA space will be reserved, but not the corresponding bytes in the CONST space. The workaround is to specify the third (“missing”, so to speak) address space sharing:

```
-U(CODE)1000-1FFF=(CONST)30000-30FFF
```

-V `-V(type)name[,align]`

where the parameters are as follows:

type	Specifies the memory type for all segments placed into the relocation area, if applicable for the target processor. If omitted it defaults to UNTYPED.
name	The name you give the area you are defining.
align	The minimum power of two alignment of the relocation area. For instance, a value of 2 means that the area will always be placed at an address that is an even multiple of 4 bytes. This value must be *at least* as high as that of any segment that will be placed into the relocation area, but preferably higher.

The -V option specifies a relocation area to place memory segments in.

Relocation areas are a way of partitioning the set of segments in such a way that a loader can place them in different parts of memory. Each relocation area has a start address that is assigned a value at load time.

When producing non-relocatable output, XLINK assigns addresses to all symbols and segment parts. When producing *relocatable* output, however, each symbol and segment part can instead be assigned an offset from the start of a relocation area. This is then turned into a regular address at load time, when the loader determines the location of each relocation area.

Relocation areas can be used instead of segment types in segment placement commands (-Z, -P).

Example

```
// Declare relocation areas for code, constants and data.
-V(CODE)CODE_AREA,12
-V(CONST)CONST_AREA,12
-V(DATA)DATA_AREA,12

// Place segments into the relocation areas
```

```
-Z(CODE_AREA)RCODE,CODE=0-FFFFFF
-Z(CONST_AREA)DATA_C,DATA_ID=0-FFFFFF
-Z(DATA_AREA)DATA_Z,DATA_I=0-FFFFFF
```

If the relocation areas share the same memory, this must be explicitly declared using the -U option, for example like this:

```
-U(CODE_AREA)0-FFFF=(CONST_AREA)0-FFFF
```

Note: Avoid mixing segment placement using relocation areas with placement using segment types. This would result in an executable file where parts are relocatable and parts are absolute, which is unlikely to be very useful.

-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 15: 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 16: 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[e46]`: Undefined external ...

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

`-x` `-x[e][h][i][m][n][s][o]`

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 42 for additional information.

The following modifiers are available:

Modifier	Description
`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. See *Symbol listing (-xe)*, page 18.
`h`	The list file will be in HTML format, with hyperlinks.
`i`	Includes all segment parts in a linked module in the list file, not just the segment parts that were included in the output. This makes it possible to determine exactly which entries that were not needed.
`m`	A list of all segments, local symbols, and entries (public symbols) for every module in the program. See *Module map (-xm)*, page 15.
`n`	Generates a module summary. See *Module summary (-xn)*, page 19.
`s`	A list of all the segments in dump order. See *Segment map (-xs)*, page 17.

Table 17: Cross-reference options

Modifier	Description
`o`	If the compiler uses static overlay, this modifier includes a listing of the static overlay system in the list file. See *Static overlay system map (-xo)*, page 20.

Table 17: 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 linker category in the IAR Embedded Workbench IDE.

-Y `-Y[`*`char`*`]`

Use `-Y` to select enhancements available for some output formats. The affected formats are `PENTICA`, `MPDS-SYMB`, `AOMF8051`, `INTEL-STANDARD`, `MPDS-CODE`, `DEBUG`, and `INTEL-EXTENDED`.

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

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

-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 linker category in the IAR Embedded Workbench IDE.

-Z `-Z[@][(type)]segment1[|align[|]][,segment2[|align[|]], … segmentn[|align[|]]][=|#]range[,range] …`

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.

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`.
segment1, *segment2*, … *segmentn*	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.
align	Increases the alignment of the segment, see *Specifying the alignment of a segment*, page 60.

= 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.
	start:+*size*	The range starting at *start* with the length *size*.
	[*start*:+*size*]**count*+*offset*	Specifies *count* ranges, where the first begins at *start* and has the length *size*, the next one begins at *start*+ *offset* and has the same length, and so on. The +*offset* part is optional, and defaults to the length of the range.
	[*start*:+*size*]/*pagesize*	Specifies the entire range beginning at *start* and with the length *size*, divided into pages of size and alignment *pagesize*. *Note*: The beginning and end of the range do not have to coincide with a page boundary.

Placement into far memory (the FAR, FARCODE, and 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.

`1000:+1000`	One range from `1000` to `1FFF`.
`[1000:+1000]*3+2000`	Three ranges: `1000-1FFF`,`3000-3FFF`,`5000-5FFF`.

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. If you want, you can put `0x` before the number to make it extra clear that it is hexadecimal, like this: `0x4FFF`.

SPECIFYING THE ALIGNMENT OF A SEGMENT

If a segment is placed using the `-Z` placement command, you can increase its alignment.

```
-Z[(type)]segment1[|align[|]][,segment2...][=ranges]
```

`align` can be any integer in the range `0–31`

`align` is treated as a decimal number (XLINK uses hexadecimal notation by default, so this is an exception). *`align`* does not specify the desired alignment in bytes, but the number of bits that are forced to zero, starting from the least significant bit of the address. The alignment thus becomes 2 raised to the power of *`align`*, so `0` means no alignment (or 1-byte aligned), `1` means 2-byte aligned, `2` means 4-byte aligned, and so on. XLINK reports alignment in the segment map part of linker list files in this way.

Examples

```
-Z(CODE)MY_ALIGNED_CODE|2=ROMSTART-ROMEND
```

This aligns the start of the segment `MY_ALIGNED_CODE` to be 4-byte aligned.

```
-Z(DATA)MY_ALIGNED_DATA|8,MY_OTHER_DATA=RAMSTART-RAMEND
```

This aligns the start of the segment `MY_ALIGNED_DATA` to be 256-byte aligned. The alignment of the `MY_OTHER_DATA` segment is not affected.

This option has no effect if the specified alignment is less than or equal to the natural alignment of the segment.

If *`align`* is surrounded by vertical bars on both sides (like `|2|`), the size of the segment will become a multiple of the segment's alignment in addition to the segment getting the alignment set by *`align`*.

Example

```
-Z(CODE)ALIGNED_CODE|2|,OTHER_ALIGNED|3,MORE_CODE=ROMSTART-ROMEND
```

This will result in `ALIGNED_CODE` becoming 4-byte aligned, and its size will be a multiple of 4. `OTHER_ALIGNED` becomes 8-byte aligned, but its size is not affected. `MORE_CODE` is not affected by the alignment of the others.

-z -z[a][b][o][p][s]

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

The option has these modifiers:

a	Ignore overlapping absolute entries
b	Ignore overlaps for bit areas
o	Check overlaps for bit areas
p	Check overlaps for SFR areas
s	Ignore overlaps for the SFR area

-za suppresses overlap errors between absolute entries. This is useful if you, for example, have several absolutely placed variables on the same address. Note that -za only ignores overlaps where both entries are absolutely placed.

All overlaps are reported by default. You can specify -zb and -zs to ignore overlaps to the bit and SFR area respectively.

For the 8051 processor, only overlaps that do not involve bit segments or SFRs are reported. You can specify -zo and -zp to report overlaps.

Using the -zs option requires that the used processor has a dedicated SFR area that XLINK has been made aware of. The only processor that has a dedicated SFR area for these purposes is the 8051. Using the -zs option for any other processor will generate warning 68 but otherwise have no effect.

Use -zb to suppress error 24 (segment overlap) for segment overlaps where at least one of the involved segments is a bit segment.

This option is identical to the **Segment overlap warnings** option in the linker category in the IAR Embedded Workbench IDE.

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	Filename extension	Address translation support	Addressing capability
AOMF8051	binary	from CPU	Yes	16 bits
AOMF8096	binary	from CPU	Yes	16 bits
AOMF80196	binary	from CPU	Yes	32 bits
AOMF80251	binary	from CPU	No	32 bits
ASHLING	binary	none	Yes	16 bits
ASHLING-6301	binary	from CPU	Yes	16 bits
ASHLING-64180	binary	from CPU	Yes	16 bits
ASHLING-6801	binary	from CPU	Yes	16 bits
ASHLING-8080	binary	from CPU	Yes	16 bits
ASHLING-8085	binary	from CPU	Yes	16 bits
ASHLING-Z80	binary	from CPU	Yes	16 bits
DEBUG (UBROF)	binary	dbg	No	32 bits
ELF*	binary	elf	Yes	32 bits
EXTENDED-TEKHEX	ASCII	from CPU	Yes	32 bits
HP-CODE	binary	x	Yes	32 bits
HP-SYMB	binary	l	Yes	32 bits
IEEE695*	binary	695	No	32 bits
INTEL-EXTENDED	ASCII	from CPU	Yes	32 bits
INTEL-STANDARD	ASCII	from CPU	Yes	16 bits
MILLENIUM (Tektronix)	ASCII	from CPU	Yes	16 bits
MOTOROLA**	ASCII	from CPU	Yes	32 bits
MOTOROLA-S19**	ASCII	from CPU	Yes	16 bits
MOTOROLA-S28**	ASCII	from CPU	Yes	32 bits
MOTOROLA-S37**	ASCII	from CPU	Yes	32 bits

Table 18: XLINK formats generating a single output file

Format	Type	Filename extension	Address translation support	Addressing capability
MPDS-CODE	binary	tsk	Yes	32 bits
MPDS-SYMB	binary	sym	Yes	32 bits
MSD	ASCII	sym	No	16 bits
MSP430_TXT	ASCII	txt	No	16 bits
NEC-SYMBOLIC	ASCII	sym	No	16 bits
NEC2-SYMBOLIC	ASCII	sym	No	16 bits
NEC78K-SYMBOLIC	ASCII	sym	No	16 bits
PENTICA-A	ASCII	sym	Yes	32 bits
PENTICA-B	ASCII	sym	Yes	32 bits
PENTICA-C	ASCII	sym	Yes	32 bits
PENTICA-D	ASCII	sym	Yes	32 bits
RAW-BINARY†	binary	bin	Yes	32 bits
RCA	ASCII	from CPU	Yes	16 bits
SIMPLE	binary	raw	No	32 bits
SIMPLE-CODE	binary	sim	No	32 bits
SYMBOLIC	ASCII	from CPU	Yes	32 bits
SYSROF	binary	abs	No	32 bits
TEKTRONIX (Millenium)	ASCII	hex	Yes	16 bits
TI7000 (TMS7000)	ASCII	from CPU	Yes	16 bits
TYPED	ASCII	from CPU	Yes	32 bits
UBROF††	binary	dbg	No	32 bits
UBROF5	binary	dbg	No	32 bits
UBROF6	binary	dbg	No	32 bits
UBROF7	binary	dbg	No	32 bits
UBROF8	binary	dbg	No	32 bits
UBROF9	binary	dbg	No	32 bits
UBROF10	binary	dbg	No	32 bits
XCOFF78k*	binary	lnk	No	32 bits
ZAX	ASCII	from CPU	Yes	32 bits

Table 18: XLINK formats generating a single output file (Continued)

* 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.

† RAW-BINARY is a binary image format. It contains no header, starting point, or address information, only pure binary data. The first byte of the file is the first byte in the application. A .bin file contains all bytes between the first and the last byte in the application, including any and all gaps. Note that there is no way to identify the entry address of the application from the contents of the file. This information must tracked of in some other way, for instance, in the filename. To link raw binary files with your application, see --*image_input*, page 38.

†† 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.

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 67.

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 35.

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 Systems 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	C++ object names. Added base types. Typedefs used in the actual types. C++ types: references and pointers to members. Class members. Target defined base types.
9	Call frame information. Function call step points. Inlined function instances.
10	C++ template information.

Table 19: 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	ann	MOTOROLA	obj
DEBUG-INTEL-EXT	DEBUG	ann	INTEL-EXT	hex
DEBUG-INTEL-STD	DEBUG	ann	INTEL-STD	hex
HP	HP-CODE	x	HP-SYMB	l

Table 20: XLINK formats generating two output files

Format	Code format	Ext.	Symbolic format	Ext.
MPDS	MPDS-CODE	tsk	MPDS-SYMB	sym
MPDS-I	INTEL-STANDARD	hex	MPDS-SYMB	sym
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	NEC78K-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 20: 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	YO	Symbols as *module*:*symbolname*.
and MPDS-SYMB	Y1	Labels and lines as *module:symbolname*.
	Y2	Lines as *module:symbolname*.

Table 21: XLINK output format variants

Output format	Option	Description
AOMF8051	Y0	Extra type of information for Hitex.
	Y1	This non-standard extension of the format can be specified to make XLINK use the SEGID field to contain the bank number (0x0–0xFF) of addresses greater than 0xFFFF. If the option is not used, the SEGID field will always be 0.
INTEL-STANDARD	Y0	End only with :00000001FF.
	Y1	End with PGMENTRY, else :0000001FF.
MPDS-CODE	Y0	Fill with 0xFF instead.
DEBUG, -r	Y#	Old UBROF version.
INTEL-EXTENDED	Y0	20-bit segmented addresses
	Y1	32 bit linear addresses
	Y2	32 bit linear addresses with no entry point
	Y3	20-bit segmented addresses with no entry point

Table 21: 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 68), ELF (see page 70), and XCOFF78K (see page 72).

IEEE695

For IEEE695 the available format modifier flags are:

Modifier	Description
-yd	Do not emit any #define constant records. This can sometimes drastically reduce the size of the output file.
-yg	Output globally visible types in a BB2 block at the beginning of the output file.
-yl	Output the globally visible types in a BB1 block at the beginning of each module in the output file.
-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.
-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).

Table 22: IEEE695 format modifier flags

Modifier	Description
`-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.
`-yv`	Use the *variable life time* support in IEEE-695 to output more accurate debug information for variables whose location vary.
`-ys`	Output IEEE-695 *stack adjust* records to indicate the offset from the stack pointer of a virtual frame pointer.
`-ya`	Output information about module local symbols in `BB10` (assembler level) blocks as well as in the `BB3` (high level) blocks, if any.
`-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 22: IEEE695 format modifier flags (Continued)

The following table shows the recommended IEEE695 format variant modifiers for some specific debuggers:

Processor family	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`
R32C	PD30, PD308, PD77, PD100 debuggers	`-ylbm`
T900	Toshiba RTE900 m25	`-ygbe`
T900	Toshiba RTE900 m15	`-ygbed`

Table 23: IEEE695 format variant modifiers for specific debuggers

ELF

For ELF the available format modifier flags are:

Modifier	Description
`-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.
`-yb`	Suppresses the generation of the `.debug_pubnames` section in the output file.
`-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.
`-yf`	Prevents the output of a `.debug_frame` section (DWARF call frame information). Note that a `.debug_frame` section is only generated if enough information is present in the linker input files.
`-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.
`-yn`	Outputs an ELF/DWARF file without debug information.
`-yo`	Generates DWARF call frame information sections that use non-factored CFA offsets instead of factored ones. Information about this will be included in the `.note.iar` section.
`-yp`	Outputs one ELF program section for each segment, instead of one section for all segments combined.
`-yr`	When this option is specified XLINK produces a relocatable executable ELF file, a file that is both executable but also contains relocation directives to make it possible to execute the image at an address other than that at which it was linked. You also need a compiler that supports relocation (consult your compiler reference guide if in doubt). *Note:* To be able to use relocatable output you also need an ELF-reader capable of reading relocatable ELF files and placing them in memory.
`-ys`	Normally, global debug information 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). Information about this will be included in the `.note.iar` section.

Table 24: ELF format modifier flags

Modifier	Description
-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.
-yw	Specify the `-yw` format variant modifier to suppress the `.debug_aranges` section in the output file. This section contains information about which addresses that a compilation unit places bytes at.
-yx	Strips the file path of all path information so the reference is only a filename, `C:\MySource\MyProject\MyFile.c` and `/home/myuser/mysource/myproject/MyFile.c` would both become references to `MyFile.c`.

Table 24: 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 following table shows the recommended ELF format variant modifiers for some specific debuggers:

Processor family	Debugger	Format variant modifier
ARM	Any ELF/DWARF debugger	-yas
H8	Renesas HEW	-yspcb
M16C	Mitsubishi PD30	-yspc
M32C	Mitsubishi KD30	-yspc

Table 25: ELF format variant modifiers for specific debuggers

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®, ColdFire®, H8, M16C, MC80, M32C, R32C, RX, SH, and V850 products.

XCOFF78K

For `XCOFF78K` the available format modifier flags are:

Modifier	Description
`-ys`	Truncates names. Use this modifier flag to truncate 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 from source file references. Use this modifier flag to strip source file paths from source file references, if there are any, 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 Systems compilers currently work these include all `#define` constants as well as all `SFR`s. Use this modifier flag to have them included.
`-yl`	Hobbles line number info. When outputting debug information, use this modifier flag to ignore any source file line number references that are not in a strictly increasing order within a function.
`-yn`	Sorts line numbers in ascending order. Normally, XLINK will output the debug information for each function in ascending *address* order. Some debuggers prefer to have the debug information in ascending *line number* order instead. Use this modifier flag to make XLINK produce debug information that is sorted in ascending line number order.

Table 26: 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`.

The following table shows the recommended XCOFF78K format variant modifiers for some specific debuggers:

Processor family	Debugger	Format variant modifier
78K0R	NEC ID78K0R-QB	`-ysp`

Table 27: XCOFF78K format variant modifiers for specific debuggers

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 46.

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 28: 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 34.

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 35.

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 35.

`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 49.

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 XLINK detects something that may be wrong. The code that is generated may still be correct.

XLINK ERROR MESSAGES

XLINK error messages are produced when XLINK 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 Systems Technical Support before sending the files.

- A list of the compiler/assembler and XLINK options that were used when the internal error occurred, including the linker command file. If you are using the IAR Embedded Workbench IDE, these settings are stored in the `prj`/`pew`/`ewp` and `dtp` files of your project. See the *IAR Embedded Workbench® IDE 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**
XLINK aborts immediately. Recompile or reassemble, or check the compatibility between XLINK and C compiler.

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

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

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

5 **Address type for CPU incorrect. Error encountered in module *module* (*file*)**
XLINK 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*)**
XLINK aborts immediately. Recompile or reassemble.

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

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

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

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

12 **Unable to open file *file***
XLINK 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. XLINK aborts immediately. Recompile or reassemble.

14 **Corrupt file. Local *local* defined twice in module *module* (*file*)**
XLINK 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. XLINK aborts immediately.

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

18 **Range error, *compiler/assembler_message***
Some instructions do not work unless a certain condition holds after linking. XLINK has verified that the conditions do not hold when the files are linked. For information about how to interpret the error message, see *Range errors*, page 12.

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

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

20 Corrupt file. External index out of range in module *module* (*file*)
The object file is corrupt. Contact IAR Systems Technical support.

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

22 Paragraph no. is not applicable for the wanted CPU. Tag encountered in module *module* (*file*)
XLINK 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. XLINK aborts immediately. Recompile or reassemble.

24 The absolute segment on the address *addressrange* in the module *module* (*file*) overlaps the segment *segmentname* (from module *module2*, address [*addressrange2*])
An absolute segment overlaps a relocatable segment. You must move either the absolute segment or the relocatable segment. You move an absolute segment by modifying the source code. You move relocatable segments by modifying the segment placement command.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

45 **Unknown CPU *CPU* encountered in *context***
XLINK 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***
XLINK 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*
XLINK aborts immediately. Do not use paragraph numbers in declarations.

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

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

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

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

55 Extended command line variable XLINK_ENVPAR is empty
XLINK aborts immediately.

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

57 No CPU defined
No CPU defined, either in the command line or in `XLINK_CPU`. XLINK aborts immediately.

58 No format defined
No format defined, either in the command line or in `XLINK_FORMAT`. XLINK aborts immediately.

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

61 This error message number is not used.

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

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

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

65 **Faulty replacement definition -e *replacement* definition**
Incorrect syntax. XLINK 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 assembler-language call from C where the required declaration using the `$DEFFN` assembler-language support directive is missing. The declaration is necessary to inform XLINK 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 command 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 XLINK (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 XLINK 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*

XLINK has found two or more absolute segments in `module` overlapping each other.

77 The absolute segment on the address *addressrange* in the module *module* (*file*) overlaps the absolute segment on the address *addressrange2* in the module *module2* (*file2*)

Two absolute segments overlap. You must move at least one of them. You move absolute segments by modifying the source code.

78 Absolute segment in module *module* (*file*) overlaps segment *segment*

XLINK has found an absolute segment in `module` (`file`) overlapping a relocatable segment.

79 Faulty allocation definition -a *definition*

XLINK 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 Too much object code produced (more than *number of bytes* bytes) for this package.

The size limit for this particular demo version was exceeded. Change the code so the end result is smaller, or upgrade the product.

90 Can only generate UBROF output from these files

This particular demo version can only generate UBROF output. You must use a KickStart version or a full version of IAR Embedded Workbench if you want to generate output in another format.

91 This XLINK version cannot link these files

These particular files from a demo version cannot be linked with this version of XLINK. Download a more recent version of XLINK.

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 XLINK when producing UBROF 5 output.

96 This error message number is not used.

97 This error message number is not used.

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 by XLINK did not manage to fit all the segments.

105 Recursion not allowed for this system. One recursive function is *functionname*.
The runtime 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
XLINK 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 Systems 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: 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 73 for more information.

134 The left and right address ranges do not cover the same number of bytes: *range1 range2*

The left and right address ranges of this command line option must cover exactly the same number of bytes.

135 A module in the file *file* has an empty module name, which is not supported in the *format* output format.

This output format cannot handle empty module names. Avoid this error by giving the module a name when you compile the source file.

136 The output format '*format*' does not support the use of relocation areas (`-V` option). Did you forget a format modifier flag?

This output format does not support relocatable output. Either the option `-y` was specified without the appropriate format modifier flag, or else the output format does not support relocatable output at all.

137 Duplicate relocation area: *relocArea1 relocarea2*

A relocation area was defined twice. Each relocation area needs a unique identifier.

138 Module *module* (*file*) contains operations that cannot be used with relocation areas: *error text*

Somewhere in the module an address (relocation area + offset) is used as if it were an absolute address. Since relocation areas usually are aligned, this is not always an error. Parts of the address could be acceptable to use.

Possible causes for this are:

- The module was compiled or assembled with a compiler or assembler that does not support relocatable output (consult your compiler reference guide if in doubt). Old IAR Systems compilers or assemblers perform checks in ways that can trigger this error (relocatable output will not work with old IAR Systems compilers).
- The alignment of your relocation area is too small. See *-V*, page 54, for details.
- If the module contains handwritten assembler code, it is possible that it uses some unknown expression that causes this error.

If the module was compiled with a modern compiler, your relocation areas has a sufficient alignment, and you get this message, contact IAR Systems Technical Support.

139 Module *module* (*file*) uses relocations (*relocation*) in ways that are not supported by the *format* output format.

The object file contains a relocation that cannot be represented in this output format. This can be the result of assembler code that uses an instruction format which is not supported by the relocation directives in this output format.

140 The range declaration used in *range declaration* is illegal since *start* > *end*.

A range must have a positive size; the end address cannot be lower than the start address.

141 The SPLIT- keyword in the packed segment placement command placement command is illegal, SPLIT- is only allowed in sequential placement commands (-Z).

Only the `-Z` placement option can use the modifier `SPLIT-`. Either use `-Z` or remove the `SPLIT-` modifier.

142 Entries included in PUBWEAK/PUBLIC resolution must be in a named segment (RSEG or ASEGN). Discovered when resolving the PUBWEAK entry in module *module* against the PUBLIC entry in module *module*.

All symbols involved the `PUBWEAK`/`PUBLIC` resolution must be placed in segments using either the `RSEG` or the `ASEGN` directive. Locate the assembler source code that defines the involved symbol in an absolute segment—using the `ASEG` directive—and replace it with a segment definition using the `ASEGN` directive.

See the *IAR Assembler Reference Guide* for information about the `ASEG` and `ASEGN` directives.

143 There is more than one PUBWEAK definition in the segment part *segment part description*.

`PUBWEAK` definitions must be perfectly interchangeable. Segment parts with multiple `PUBWEAK` definitions cannot not always be interchanged with other definitions of the same symbols.

144 The conditional reference at offset *offset* in segment *segment* could not use its definition of last resort, the entry in segment *segment*.

In order for XLINK to be able to optimize the use of relay functions, each module must supply relay functions that can be used by every call site in that module. This error occurs when that precondition is not met. The distance between the reference and the definition might be too large, or the definition might be unsuitable because it is in the wrong processor mode, or for some other reason.

If this occurs for a module produced by a compiler (as opposed to in assembler code), this is an indication of a problem in either the compiler or the linker. To test if the problem is in the linker, try linking with the option **Relay Function Optimization** disabled (`-q`).

145 The banked segment *segment* contains segment parts that have properties that are unsafe when placed with -b (banked segment placement). Use -P (packed segment placement) instead.

The segment contains at least one segment part with a property that XLINK might be unable to handle when the segment is placed using the placement option `-b`. Use the placement option `-P` instead.

146 Type conflict for external/entry "*entry1*", in module *module1* against external/entry *entry2* in module *module2* — if objects or functions are declared more than once, they shall have compatible declarations. (MISRA C rule 26)

147 External "*external*" is declared in "*file1*" and in "*file2*" — external objects should not be declared in more than one file " ". (MISRA C rule 27)

148 The names "*name1*" and "*name2*" differ only in characters beyond position 31 — identifiers (internal and external) shall not rely on significance of more than 31 characters. (MISRA C rule 11)

149 The symbol "*symbol*" in module *module* (*file*) is public but is only needed by code in the same module — all declarations at file scope should be static where possible. (MISRA C rule 23)

150 The stack depth for the call tree with root *root* is too large, *number* bytes.

The call tree uses more than the allowed number of stack bytes. You must either increase the maximum allowed call depth, or decrease the depth of the call tree.

151 Internal consistency check failure, "*error description*".

An internal consistency check failed. This is an internal error, but you can force XLINK to generate output using the `-B` option.

152 The input file '*file*' could not be found.

The input file could not be found. Check the include path.

153 The input file '*file*' has several forced properties which are mutually exclusive.

The input file has both the conditional and forced load properties. Locate the mutually exclusive `-A` and `-C` options and remove the filename from one of them.

154 The increment argument to -K for the segment *SEGMENTNAME* resulted in an invalid (negative or above 0xFFFFFFFF) address.

The duplication command for `SEGMENTNAME` results in at least one duplicated segment that has an address below `0` or above `0xFFFFFFFF`. You must either modify the `-K` command (the difference or the number of duplications) or move the segment to another address, to prevent this from happening.

155 The program uses static overlay, this is not allowed in the basemap format.

Your application contains 1 or more bytes stored in a static overlay frame. Static overlay is currently not supported in the basemap output format.

156 **Negative addresses are not allowed. The range declaration used in *range description* is illegal as *range start* is negative. Check the range specification for superfluous parentheses, (START–END) is an expression, not a range, START–END is a range.**

The range declaration has a negative start value. This is not allowed.Check the range specification for superfluous parentheses and check that the value of `START` and `END` are valid and that `START<=END`.

157 **Debug information must be disabled for the '*cpu*' processor in the '*format*' output format. Use the appropriate -y option to suppress the generation of debug information.**

Output for the device you have selected is still experimental in this format. No debug information can currently be generated and the generation must therefore be disabled.

158 **The directory name *directory* is not valid.**

The specified name is not a valid directory name on this system.

159 **The file name *file* is not valid.**

The specified name is not a valid filename on this system.

160 **No valid license found for this product. *Information from the license management system.***

No valid license was found for at least one module that needed a license. You either do not have the required license or XLINK was unable to contact the license server.

161 **The checksum command defined in *checksum command* specifies an initial value that does not fit in the size of the checksum.**

The initial value specified is too large for the size of the checksum. Use a smaller initial value or increase the size of the checksum.

162 **Alignment specification (|align[|]) is not allowed for segment names here: *use of segment name***

Alignment specification on segments are only allowed in a sequential segment placement command (`-Z`).

163 **The command line symbol "*symbol*" in *command line option* is not defined.**

The indicated command line option contains a symbol with an undefined value. Define the symbol (`-Dsymbol=value`) or use a symbol that is defined.

164 **The option *command line option* contains neither a number nor a command line symbol.**

The indicated command line option contains characters that are neither part of a number (`0–9` and `A–F`) nor valid in a symbol name, in a place where a number or a symbol was expected.

165 A segment definition in *segment placement command* uses an alignment argument that is larger than the currently supported maximum (31).

XLINK currently only supports alignments up to and including 2^{31}.
Remember that the alignment argument in the segment placement command is the number of bits in the address that are forced to zero, not the byte alignment. `2` results in a 4-byte aligned address, `3` in an 8-byte aligned address, and so on. See *Specifying the alignment of a segment*, page 60.

166 In the chosen byte order for the *processor* processor, you must specify the code fill option (-hc) or the range fill option (-H without any -h option).

For this particular processor, in this particular byte order, the **Code fill** option must be specified (because of the special requirements of bi-endian code).

167 Generation of bi-endian output files is not supported for the '*output format*' output format.

This output format does not currently support generation of bi-endian files. You must choose another format, or use the processor in either big-endian or little-endian mode.

168 Alignment error, segment part *segment part number* ("*symbol*") in the module '*module*' (*file*) that generated the bi-endian content on address *address* does not have the required alignment.

Bi-endian code must be generated in such a way that every word is either code or non-code. In the segment part specified above, this requirement is not met. If the object file was generated by the compiler, this is probably a compiler problem. If the object file was generated by the assembler, the code probably needs to be aligned and/or padded.

169 Processor specific code fill (-hc) requires all ranges to be closed. The placement command "*segment placement command*" contains an open range.

All ranges must be closed when you use this option. Use either `START-END` or `START:+SIZE` to specify a closed range.

170 The segment "*segment*" that is used in a checksum command has not been defined.

The specified segment does not exist. Define it using the option `-Z` or use a different segment in the checksum command.

171 The segment "*segment*" that is used in a checksum command is a packed segment.

Segment names used in a checksum command must be sequentially placed. Place the segment using the option `-Z` or use an explicit address range (like `0x200–0x37F`) in the checksum command.

172 Output for the *processor* processor in this byte order will use bi-endian code segments. This requires the code segments to be aligned (both start and size) to *alignment* bytes. The following segments do not have the required alignment: *list of segments*

Bi-endian code that is not properly aligned will not work. Align the listed segments (see *Specifying the alignment of a segment*, page 60) or make sure that the code is aligned in the compiler/assembler.

173 Unable to locate the dll *dll*

XLINK was unable to find the DLL. Make sure that the DLL is present in the expected location and that the path to the location is made available through use of the `-I` option (or use an absolute path).

174 The dll *dll* reports a problem. It will not be able to descramble the file *file*

The DLL encountered a problem while attempting to descramble the file. Some possible explanations are:

- Problems with the file, it might be corrupt.
- Problems with the DLL itself, it might be corrupt.
- Problems with the license checking system in the DLL (if any).

Warning messages

The following section lists the IAR XLINK 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

XLINK 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*

XLINK has found two relocatable segments overlapping each other. Check the segment placement option parameters.

19 Absolute segments overlaps in module *module* (*file*)

XLINK has found two or more absolute segments in module `module` overlapping each other.

20 The absolute segment on the address *addressrange* in the module *module* (*file*) overlaps the segment *segmentname* (from module *module2*, address [*addressrange2*])

An absolute segment overlaps a relocatable segment. You must move either the absolute segment or the relocatable segment. You move an absolute segment by modifying the source code. You move relocatable segments by modifying the segment placement command.

21 The absolute segment on the address *addressrange* in the module *module* (*file*) overlaps the absolute segment on the address *addressrange2* in the module *module2* (*file2*)

Two absolute segments overlap. You must move at least one of them. You move absolute segments by modifying the source code.

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 used output format, or in the debug information available, XLINK cannot produce correct debug output for this application. This only affects the debug information; the generated code remains the same as in an output format where the debug information can be expressed. 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. XLINK 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.

57 The file *filename* is empty and will be ignored.

The file is completely empty (0 bytes). It is not a valid UBROF file, but some IAR Systems assemblers generate completely empty files instead of a valid UBROF file with no content.

This file will be ignored. If the file was not generated by an IAR Systems assembler, you should find out why it is empty.

58 The name *name* was too long (more than *number* characters) and has been truncated to fit the chosen output format. This warning is only issued once.

Normally, this will not affect debugging to any great extent, but if two or more long names are truncated to the same 255-character string, the code can become harder to debug.

The most common case where long names occur is when C++ names are flattened into simple strings, which occurs when translating into UBROF version 7 or earlier, or into other debug formats with limited symbol name length.

59 Too many COFF format line number records (*number*) needed. All in excess of 65535 will not be accessible.

There are too many line number records in one COFF section. This can make the application much harder to debug if the original number of records greatly exceeds 65535.

One way to avoid this is to put code in more than one segment, because one COFF section is output for each segment.

This problem is most likely to occur in the MPLAB debugger for the PIC processor family, because it needs one line number record for each instruction in the code.

60 The entry point label "*label*" was not found in any input file. The image will not have an entry point.

The chosen entry point label could not be found in any input file. Choose an entry point that exists in the program or make sure that the file that contains the entry point is included in the input files.

61 The '*format*' output format is not supported for this cpu.

Support for the chosen output format is experimental for this cpu.

62 The struct "*struct*" is too large for the '*format*' format, debug information will only be available for the first *maximum size* bytes.

The program contains a class, struct, or union that is too large to represent in the chosen debug format. Debug information will be generated for as many bytes as the format can represent.

63 No debug information will be generated for the function "*function* in the module "*module*" as no debug information could be found.

This likely because of a rename entry operation in the IAR XLIB Librarian.

64 The address space used in the command *segment placement command* is incompatible with the address space of the ranges *ranges* that were inherited from previous placements. Address ranges can only be inherited from compatible address spaces.

Addresses should not be inherited from previous placement commands, if those previous commands placed segments in an incompatible address space. This technique was used in some older IAR tools to make sure that segments placed in overlapping address spaces did not overlap each other. Use the `-U` option instead to prevent this.

65 There are both MULTWEAK and PUBWEAK definitions for the symbol named *name*. This does not work in the general case. PUBWEAK definitions occur in the module(s) *modules*. MULTWEAK definitions occur in the module(s) *modules*.

`MULTWEAK` definitions were introduced to be used by ARM/Thumb Relay Function Optimization in the ARM® IAR C/C++ Compiler v3.41. `PUBWEAK` definitions were used for the same purpose in earlier versions of the ARM IAR C/C++ Compiler. In order to avoid this problem, ensure that all modules are built for use with the same Relay Function model.

66 There is a gap between the addresses *address1* and *address2*. This gap of *gap_size* bytes will be padded with zeroes. Raw-binary might not be the format you want for this particular image.

There is a huge "hole" in the image. This might result in an unnecessarily large file. A format that uses address records (like Intel-extended, Motorola or simple-code) might be a better choice for this particular image.

67 Using "-r" causes XLINK to select modules that are adapted for use with the C-SPY Debugger. This affects all output files, including those generated by -O.

The linker command line option `-r` has two effects. It causes XLINK to select modules from the IAR standard library designed to work with the IAR C-SPY® Debugger and it makes XLINK use the IAR UBROF object format for its main output file. The first of these effects also changes the contents of any extra output files produced by the use of the `-O` command line option.

If you need extra output files not meant for use with the IAR C-SPY Debugger, you need to run XLINK separately, without the `-r` command line option.

68 The option to ignore overlaps in SFR areas has been specified but the '*processor*' processor does not have an SFR area. The option has no effect for this processor.

The processor you are using does not have a dedicated SFR area (an address range in an address space that can only contain SFRs). You cannot use the `-zs` option to suppress segment overlap errors when using this processor.

69 Address translation (-M, -b# or -b@) has no effect on the output format '*format*'. The output file will be generated but no address translation will be performed.

Address translation is not supported for the indicated output format. Output files in that output format will be generated without address translation. However, if there are also output files in a format for which address translation is supported, those files will use translated addresses.

70 The segment "*segment*" on address *address* overlaps previous content in the raw-binary output file. The previous content will be overwritten.

Your application contains at least one overlap between segments with content. Locate and correct those overlaps.

71 This warning message number is not used.

72 The *format* output format does not support line numbers above *number.* All line numbers above this limit will be set to *number.* Source information for such functions will not be available. This will affect the following functions: *list of function names, their segment part numbers, the module and file where they were defined*

The chosen output format cannot represent line numbers as high as the ones in the specified files. To debug the specified functions on C level, you must take one of these actions:

- Use a different output format

- Edit the source code file(s) so that no function that you wish to debug resides on a higher line number than the limit (move the function or split the file into two or more smaller files)

Even if you take neither action, assembly level debugging and variable information is still available.

73 **Total number of warnings for unsupported line numbers: *number of warnings***

This warning is connected to warning 72. It is, for example, useful if you suppress warning 72 (because you do not want to list all functions on line numbers that are too high every time you link), but you still want to know if the number of warnings change.

74 **The checksum polynomial *polynomial* is unsuitable for use with the bytewise initial value #*initial value* as the polynomial can not always generate a bitwise equivalent for the bytewise initial value. Use a bitwise initial value or use a polynomial with the least significant bit set.**

The specified polynomial does not have its least significant bit set. When the least significant bit is not set, it is not always possible to convert a bytewise initial value to a bitwise initial value. This has no real effect on the checksum that XLINK generates, but it can matter if the verification step uses bitwise initial values. Because some bytewise initial values lack a corresponding bitwise initial value for this polynomial, it might be impossible to get the checksums to match.

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 Systems library tools
- The IAR XAR Library Builder
- XAR diagnostics
- IAR XLIB Librarian options
- XLIB diagnostics.

Introduction to the IAR Systems library tools

This chapter describes XAR Library Builder and IAR XLIB Librarian—the IAR Systems 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/C++ programs

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

Most C or C++ programmers will use one or both of the IAR Systems 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, C++, or assembler modules to the standard library file so they will always be available whenever a C/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 IAR DLIB 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 assembler 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*.

The IAR XAR Library Builder

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 29: 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 30: 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® IDE.

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 acquire 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 occurred while writing to library file**
An error occurred 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 occurred while reading from *filename***
An error occurred while XAR was reading the file.

10 **Error occurred while closing *filename***
An error occurred 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.

14 -x option requires an argument
You must specify an argument for the `-x` option.

15 A file name in the file *filename* exceeds the maximum filename length of *number* characters.
A filename in the extended command line file is too long. The only recognized delimiter in the input file is the `newline` character, everything else is interpreted as a part of the filename.

IAR XLIB Librarian 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 31: 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 32: 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 33: 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 34: 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 34: 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 35: 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.
RENAME-MODULE	Renames one or more modules.
RENAME-SEGMENT	Renames one or more segments.
REPLACE-MODULES	Updates executable code.

Table 35: XLIB options summary (Continued)

Note: There are some XLIB options that do not work with the output from modern IAR Systems C/C++ compilers or assemblers. See *Summary of XLIB options for older UBROF versions*, page 130.

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.r`*`nn`*:

```
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 119.

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 119.

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 119.

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 119.

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 119.

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 119.

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 119.

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 119.

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
      Repl   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
```

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
```

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
```

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 runtime 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 Systems C/C++ 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 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.

Table 36: 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 Systems C/C++ 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 Systems C/C++ 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 Systems C/C++ 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 ***Filename* is write protected**
The file *`filename`* 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

-A (XLINK option) 31
-a (XLINK option) 31
address range check, disabling 51
address space
restricting output to one 73
sharing 53
address translation 10, 43
addresses, mapping logical to physical 42
alignment for checksums 39
alignment, of a segment 60
allocation, segment types 10
Always generate output (XLINK option) 32, 79
AOMF80196 (linker output format) 63
AOMF80251 (linker output format) 63
AOMF8051 (linker output format) 63, 68
AOMF8096 (linker output format) 63
ARM compiler, disabling relay function optimization 51
ASCII format, of object code listing 127
ASHLING (linker output format) 63
ASHLING-Z80 (linker output format) 63
ASHLING-6301 (linker output format) 63
ASHLING-64180 (linker output format) 63
ASHLING-6801 (linker output format) 63
ASHLING-8080 (linker output format) 63
ASHLING-8085 (linker output format) 63
assembler directives
MODULE 111
NAME 111
assembler symbols, defining at link time 34
assumptions (programming experience) ix

B

-B (XLINK option) 32, 79
-b (XLINK option) 9, 32
banked segments
defining 32, 47
range errors suppressed for 52
binary files, linking 38
BIT (segment type) 11
bold style, in this guide xi
bytewise initial value, of checksum 40

C

-C (XLINK option) 34
-c (XLINK option) 34, 76
checksum
generating in XLINK 38
initial value of 40
mirroring initial value 40
__checksum (default label) 39
checksum calculation, included bytes 40
checksum value symbol 26
CHECKSUM (default segment name) 39
checksummed areas 22
__checksum__value (default checksum value symbol) 26
code duplication, in XLINK 41
code generation, disabling in XLINK 34
code memory, filling unused 36
CODE (segment type) 11
command file comments, including in XLIB 129
command files, debugging 123
command line options
typographic convention xi
command prompt icon, in this guide xi
comments
in XLIB command files, including 129
COMMON (segment type) 10
COMPACT-FILE (XLIB option) 121
computer style, typographic convention xi
CONST (segment type) 11
conventions, used in this guide x
copyright notice ii
CPU, defining in XLIB 122
CRC value of modules, listing 125

crc=n (checksum algorithm) . . . 39
crc16 (checksum algorithm) . . . 39
crc32 (checksum algorithm) . . . 39
cross-reference, in XLINK listing . . . 15, 56
  See also -x (XLINK option)
C++ terminology . . . x

D

-D (XLINK option) . . . 34
-d (XLINK option) . . . 34
DATA (segment type) . . . 11
data, storing locals at static locations . . . 20
debug information
  generating in XLINK . . . 52
  loss of . . . 65
DEBUG (linker output format) . . . 63, 68
DEBUG-INTEL-EXT (linker output format) . . . 66
DEBUG-INTEL-STD (linker output format) . . . 66
DEBUG-MOTOROLA (linker output format) . . . 66
default extension, setting in XLIB . . . 123
#define (XLINK option) . . . 34
DEFINE-CPU (XLIB option) . . . 122
DELETE-MODULES (XLIB options) . . . 122
diagnostics
  XAR . . . 115
  XLIB . . . 133
  XLINK . . . 79
diagnostics control, XLINK . . . 55
direct initial values . . . 27
DIRECTORY (XLIB option) . . . 122
directory, specifying in XLINK . . . 42
disclaimer . . . ii
DISPLAY-OPTIONS (XLIB option) . . . 122
document conventions . . . x
dtp (file type) . . . 80
duplicating code, in XLINK . . . 41
DWARF (linker output format) . . . 70

E

-E (XLINK option) . . . 34
-e (XLINK option) . . . 35
ECHO-INPUT (XLIB option) . . . 123
edition, of this guide . . . ii
ELF (linker output format) . . . 63, 70
entry list, XLINK . . . 18
entry point for applications, specifying in XLINK . . . 52
environment variables
  XLIB, summary of . . . 119
  XLINK . . . 75
  XLINK_COLUMNS . . . 75
  XLINK_CPU . . . 34
  XLINK_DFLTDIR . . . 37
  XLINK_ENVPAR . . . 29
  XLINK_FORMAT . . . 35, 76
  XLINK_PAGE . . . 49, 77
error messages
  range . . . 12
    suppressed . . . 51
  segment overlap . . . 12
  XAR . . . 115
  XLIB . . . 133
  XLINK . . . 79
ewp (file type) . . . 80
EXIT (XLIB option) . . . 123
experience, programming . . . ix
EXTENDED-TEKHEX (linker output format) . . . 63
EXTENSION (XLIB option) . . . 123
extension, setting default in XLIB . . . 123
EXTERN symbols, renaming in XLIB . . . 131
external symbols
  defining at link time . . . 34
  listing . . . 126
  renaming in XLIB . . . 131
  renaming in XLINK . . . 35
Extra output (XLINK option) . . . 46

F

-F (XLINK option) 35, 52, 77
-f (XLINK option) 35, 76
far memory, placing segments in 59
FAR (segment type) 11
  range errors suppressed for 51
FARC (segment type) 11
  range errors suppressed for 51
FARCODE (segment type) 11
  range errors suppressed for 51
FARCONST (segment type) 11
  range errors suppressed for 51
fast CRC 27
features, XLINK 3
FETCH-MODULES (XLIB option) 123
file types
  dtp 80
  ewp 80
  hex 47
  lst 42
  map 42
  pew 80
  prj 80
  xcl 29
filename, specifying for XLINK listing 42
Fill unused code memory (XLINK option) 36
filler bytes 36
filling ranges 36
filling unused code memory 36
Format variant (XLINK option) 67
format variant, specifying in XLINK 57
formats
  assembler object file 5
  assembler output 6
  compiler object file 5
  UBROF 5
  XLIB list file 119, 127
  XLINK listing 14
  XLINK output 63
    variants 67
functions, in XLINK 5

G

-G (XLINK option) 36
-g (XLINK option) 36
global entries 36
global type checking, disabling 36

H

-H (XLINK option) 22, 36, 38
-h (XLINK option) 36
help information, displaying in XLIB 124
HELP (XLIB option) 124
hex (file type) 47
HP (linker output format) 66
HP-CODE (linker output format) 63
HP-SYMB (linker output format) 63
HUGE (segment type) 11
  range errors suppressed for 52
HUGEC (segment type) 11
  range errors suppressed for 52
HUGECODE (segment type) 11
  range errors suppressed for 52
HUGECONST (segment type) 11
  range errors suppressed for 52

I

-I (XLINK option) 37
IAR XAR Library Builder. *See* XAR
IAR XLIB Librarian. *See* XLIB
IAR XLINK Linker. *See* XLINK
icons, in this guide xi
IDATA (segment type) 11
IDATA0 (segment type) 11

IDATA1 (segment type) . . . 11
IEEE695 (linker output format) . . . 63
IEEE695 (XLINK output format) . . . 68
--image_input (XLINK option) . . . 38
include paths, specifying to XLINK . . . 37
indirect initial values . . . 27
initial value, of checksum . . . 40
input files and modules, XLINK . . . 6
INSERT-MODULES (XLIB option) . . . 124
instruction set of microcontroller . . . ix
INTEL-EXTENDED (linker output format) . . . 63, 68
INTEL-STANDARD (linker output format) . . . 63, 68
introduction
  MISRA C . . . 4
  XAR . . . 109
  XLIB . . . 109
  XLINK . . . 3
italic style, in this guide . . . xi

J

-J (XLINK option) . . . 22, 38

K

-K (XLINK option) . . . 9, 41

L

-L (XLINK option) . . . 42
-l (XLINK option) . . . 42
large address awareness . . . 4
librarian. *See* XLIB *or* XAR
libraries . . . 109
  See also library modules
  building . . . 114
  file size, reducing . . . 121
  module type, changing . . . 128
  using with assembler programs . . . 110
  using with C programs . . . 110
library modules
  adding . . . 123
  building and managing . . . 109
  inserting . . . 124
  loading . . . 6–7, 34
  removing . . . 122
lightbulb icon, in this guide . . . xi
Lines/page (XLINK option) . . . 49
linker command file . . . 29
  specifying . . . 35
Linker command file (XLINK option) . . . 36
linker. *See* XLINK
linking . . . 4
linking protected files . . . 4
list file formats
  XLIB . . . 119
  XLINK . . . 14
List (XLINK option) . . . 42, 57
listings
  generating in XLINK . . . 42
  lines per page . . . 49
LIST-ALL-SYMBOLS (XLIB option) . . . 124
LIST-CRC (XLIB option) . . . 125
LIST-DATE-STAMPS (XLIB option) . . . 126
LIST-ENTRIES (XLIB option) . . . 126
LIST-EXTERNALS (XLIB option) . . . 126
LIST-MODULES (XLIB option) . . . 127
LIST-OBJECT-CODE (XLIB option) . . . 127
LIST-SEGMENTS (XLIB option) . . . 128
local data, storing at static locations . . . 20
local symbols, ignoring . . . 45
lst (file type) . . . 42

M

-M (XLINK option) . . . 9, 42
MAKE-LIBRARY (XLIB option) . . . 128
MAKE-PROGRAM (XLIB option) . . . 128

map (file type). 42
memory
  code, filling unused 36
  far, placing segments in 59
  segment types. 11
microcontroller instruction set ix
MILLENIUM (linker output format) 63
mirroring. 28
mirroring the initial value, of checksum 40
MISRA C 4
  checking for violations against. 44
  logging violations against 45
MISRA C (XLINK options) 45
--misrac (XLINK option) 44
--misrac_verbose (XLINK option) 45
module summary, XLINK. 19
MODULE (assembler directive) 111
modules
  adding to library. 123
  changing to program type 128
  generation date, listing 126
  inserting in library 124
  library, loading in XLINK 6–7
  listing 127
    CRC value. 125
    EXTERN symbols 126
    PUBLIC symbols 126
    segments 128
    symbols in. 124
  loading as library 34
  loading as program. 31
  removing from library 122
  renaming 129
  replacing 130
  type, changing to library 128
MOTOROLA (linker output format). 63, 65
MOTOROLA-S19 (linker output format) 63, 65
MOTOROLA-S28 (linker output format) 63, 65
MOTOROLA-S37 (linker output format) 63, 65
MPDS (linker output format) 67
MPDS-CODE (linker output format) 64, 68
MPDS-I (linker output format) 67
MPDS-M (linker output format) 67
MPDS-SYMB (linker output format) 64, 67
MSD (linker output format) 64
MSD-I (linker output format) 67
MSD-M (linker output format) 67
MSD-T (linker output format) 67
MSP430_TXT (linker output format) 64

N

-N (XLINK option). 45
-n (XLINK option) 45
NAME (assembler directive). 111
naming conventions xi
NEAR (segment type). 11
  range errors suppressed for 51
NEARC (segment type) 11
  range errors suppressed for 51
NEARCODE (segment type)
  range errors suppressed for 51
NEARCONST (segment type) 11
  range errors suppressed for 51
NEC (linker output format). 67
NEC-SYMBOLIC (linker output format) 64
NEC2 (linker output format). 67
NEC2-SYMBOLIC (linker output format) 64
NEC78K (linker output format) 67
NEC78K-SYMBOLIC (linker output format). 64
No global type checking (XLINK option). 36
NPAGE (segment type). 11

O

-O (XLINK option). 46
-o (XLINK option) 47
-o (XAR option) 114

object code
listing in ASCII format 127
suppressing in XLINK 34
object files
displaying available 122
format 5
Old UBROF (linker output format). 65
ON-ERROR-EXIT (XLIB option) 129
option summary
XAR 113
XLIB 117
XLINK. 29
options (XLINK), setting from the command line 29
output file name (XLINK), specifying 47
output files, multiple. 46
output format
XLINK. 6, 63
specifying 35
variant, specifying. 57
Output (XLINK option) 35, 46–47, 52, 57
output, generating in XLINK also on error 32
overlap errors 12
overlay system map. 20
Override default program entry (XLINK option) 52

P

-P (XLINK option) 9, 47
-p (XLINK option) 49, 77
packed segments, defining 47, 58
parameters, storing at static locations 20
parameters, typographic convention xi
part number, of this guide ii
PENTICA-A (linker output format) 64, 67
PENTICA-AI (linker output format). 67
PENTICA-AM (linker output format) 67
PENTICA-B (linker output format) 64, 67
PENTICA-BI (linker output format). 67
PENTICA-BM (linker output format). 67
PENTICA-C (linker output format) 64, 67
PENTICA-CI (linker output format). 67
PENTICA-CM (linker output format). 67
PENTICA-D (linker output format) 64, 67
PENTICA-DI (linker output format). 67
PENTICA-DM (linker output format) 67
pew (file type). 80
prerequisites (programming experience). ix
prj (file type). 80
processor type
specifying in XLIB 122
specifying in XLINK 34
program modules
changing module type 128
loading as. 31
programming experience, required ix
PUBLIC symbols
listing. 126
renaming in XLIB 131
publication date, of this guide ii

Q

-Q (XLINK option). 49
-q (XLINK option) 51
QUIT (XLIB option). 129

R

-R (XLINK option). 51
-r (XLINK option) 52
Range checks (XLINK option) 52
range check, disabling. 51
range errors. 12
suppressed 51
ranges, filling 36
Raw binary image (XLINK option) 38
RAW-BINARY (linker output format) 64–65
RCA (linker output format). 64

reference information, typographic convention . . . xi
reflection . . . 28
registered trademarks . . . ii
RELATIVE (segment type) . . . 10
relay function optimization, disabling . . . 51
relocatable code and data, placing in memory . . . 54
relocation areas, in XLINK . . . 54
REMARK (XLIB option) . . . 129
RENAME-ENTRY (XLIB option) . . . 131
RENAME-EXTERNAL (XLIB option) . . . 131
RENAME-GLOBAL (XLIB option) . . . 131
RENAME-MODULE (XLIB option) . . . 129
RENAME-SEGMENT (XLIB option) . . . 130
REPLACE-MODULES (XLIB option) . . . 130
root, forcing . . . 45
-rt (XLINK option) . . . 52
runtime libraries, updating . . . 130

S

-S (XLINK option) . . . 52
-s (XLINK option) . . . 52
safety-critical systems, developing . . . 4
scatter loading, in XLINK . . . 49
Search paths (XLINK option) . . . 37
segment allocation . . . 9, 32, 47, 58
segment control options (XLINK) . . . 9
segment map
  including in XLINK listing . . . 56
  XLINK . . . 15–17
segment overlap errors . . . 12
  reducing . . . 61
Segment overlap warnings (XLINK option) . . . 61
segment overlaps, creating . . . 58
segment types
  allocation . . . 10
  BIT . . . 11
  CODE . . . 11
  COMMON . . . 10
  CONST . . . 11
  DATA . . . 11
  FAR . . . 11
  far memory . . . 59
  FARC . . . 11
  FARCODE . . . 11
  FARCONST . . . 11
  HUGE . . . 11
  HUGEC . . . 11
  HUGECODE . . . 11
  HUGECONST . . . 11
  IDATA . . . 11
  IDATA0 . . . 11
  IDATA1 . . . 11
  memory . . . 11
  NEAR . . . 11
  NEARC . . . 11
  NEARCONST . . . 11
  NPAGE . . . 11
  RELATIVE . . . 10
  STACK . . . 10
  UNTYPED . . . 11
  XDATA . . . 11
  ZPAGE . . . 11
segments . . . 8
  banked
    defining . . . 32, 47
    range errors suppressed for . . . 52
  copy initialization . . . 49
  listing in modules . . . 128
  packed, defining . . . 47, 58
  placing in far memory . . . 59
  renaming . . . 130
silent operation, in XLINK . . . 52
SIMPLE (linker output format) . . . 64
SIMPLE-CODE (linker output format) . . . 64
slow crc . . . 27
STACK (segment type) . . . 10
static memory, storing local data in . . . 20

static overlay system map . . . 20
static overlay, disabling . . . 31
sum (checksum algorithm) . . . 39
support, technical . . . 79
symbol listing, XLINK . . . 18
SYMBOLIC (linker output format) . . . 64
symbols
  defining at link time . . . 34
  EXTERN, listing . . . 126
  ignoring local at link time . . . 45
  in modules, listing . . . 124
  PUBLIC, listing . . . 126
  renaming EXTERN . . . 131
  renaming PUBLIC . . . 131
SYSROF (linker output format) . . . 64

T

target processor, specifying in XLINK . . . 34
target ROM, comparing to debug file . . . 26
technical support, reporting errors to . . . 79
TEKTRONIX (linker output format) . . . 64
terminal I/O, emulating . . . 52
terminology . . . x
TI7000 (linker output format) . . . 64
TMS7000 (linker output format) . . . 64
tools icon, in this guide . . . xi
trademarks . . . ii
translation, address . . . 10
type checking, disabling global . . . 36
TYPED (linker output format) . . . 64
typographic conventions . . . xi

U

-U (XLINK option) . . . 53
UBROF
  generating debug information in output file . . . 52
  object file format . . . 5
  properties . . . 64
  UBROF 7 or later . . . 9–10
  version . . . 65
    specifying . . . 65
    XLIB options dependent of . . . 130
UBROF5 (linker output format) . . . 64
UBROF6 (linker output format) . . . 64
UBROF7 (linker output format) . . . 64
UBROF8 (linker output format) . . . 64
UBROF9 (linker output format) . . . 64
UBROF10 (linker output format) . . . 64
Universal Binary Relocatable Object Format . . . 5
UNTYPED (segment type) . . . 11

V

-V (XLINK option) . . . 54
-V (XAR option) . . . 114
version
  IAR Embedded Workbench . . . ii
version, XLINK . . . ix

W

-w (XLINK option) . . . 55
warning messages, XLINK . . . 97
  controlling . . . 55
warnings icon, in this guide . . . xi

X

-x (XLINK option) . . . 15–19, 56
XAR
  basic syntax . . . 113
  differences from XLIB . . . 109
  error messages . . . 115
  introduction to . . . 109
  verbose mode . . . 114

XAR options
- summary . . . 113
- -o . . . 114
- -V . . . 114

xcl (file type) . . . 29
XCOFF78K (linker output format) . . . 72
XCOFF78k (linker output format) . . . 64
XDATA (segment type) . . . 11
XLIB
- differences from XAR . . . 109
- error messages . . . 133
- introduction to . . . 109

XLIB help information, displaying . . . 124
XLIB list file format . . . 119
XLIB options
- COMPACT-FILE . . . 121
- DEFINE-CPU . . . 122
- DELETE-MODULES . . . 122
- DIRECTORY . . . 122
- displaying . . . 122
- DISPLAY-OPTIONS . . . 122
- ECHO-INPUT . . . 123
- EXIT . . . 123
- EXTENSION . . . 123
- FETCH-MODULES . . . 123
- HELP . . . 124
- incompatible with modern compilers . . . 130
- INSERT-MODULES . . . 124
- LIST-ALL-SYMBOLS . . . 124
- LIST-CRC . . . 125
- LIST-DATE-STAMPS . . . 126
- LIST-ENTRIES . . . 126
- LIST-EXTERNALS . . . 126
- LIST-MODULES . . . 127
- LIST-OBJECT-CODE . . . 127
- LIST-SEGMENTS . . . 128
- MAKE-LIBRARY . . . 128
- MAKE-PROGRAM . . . 128
- ON-ERROR-EXIT . . . 129
- QUIT . . . 129
- REMARK . . . 129
- RENAME-ENTRY . . . 131
- RENAME-EXTERNAL . . . 131
- RENAME-GLOBAL . . . 131
- RENAME-MODULE . . . 129
- RENAME-SEGMENT . . . 130
- REPLACE-MODULES . . . 130
- summary . . . 117

XLINK options
- Always generate output . . . 32, 79
- Extra output . . . 46
- Fill unused code memory . . . 36
- Format variant . . . 67
- Lines/page . . . 49
- Linker command file . . . 36
- List . . . 42, 57
- MISRA C . . . 45
- No global type checking . . . 36
- Output . . . 35, 46–47, 52, 57
- Override default program entry . . . 52
- Range checks . . . 52
- Raw binary image . . . 38
- Search paths . . . 37
- Segment overlap warnings . . . 61
- Target processor . . . 34
- -A . . . 31
- -a . . . 31
- -B . . . 32, 79
- -b . . . 9, 32
- -C . . . 34
- -c . . . 34, 76
- -D . . . 34
- -d . . . 34
- -E . . . 34
- -e . . . 35
- -F . . . 35, 52, 77
- -f . . . 35, 76
- -G . . . 36

-g 36
-H. . . . 22, 36, 38
-h 36
-I 37
-J 22, 38
-K. . . . 9, 41
-L. . . . 42
-l 42
-M 9, 42
-N. . . . 45
-n 45
-O. . . . 46
-o 47
-P 9, 47
-p 49, 77
-Q. . . . 49
-q 51
-R. . . . 51
-r 52
-rt 52
-S 52
-s 52
-U. . . . 53
-V. . . . 54
-w. . . . 55
-x 15–19, 56
-Y. . . . 57, 67
-y 57
-Z. . . . 9, 58
-z 61
--image_input. . . . 38
--misrac 44
--misrac_verbose 45
#define 34
XLINK_COLUMNS (environment variable) 75
XLINK_CPU (environment variable) 34, 76
XLINK_DFLTDIR (environment variable). . . . 37, 76
XLINK_ENVPAR (environment variable) 29, 76
XLINK_FORMAT (environment variable). . . . 35, 76
XLINK_PAGE (environment variable) 49, 77

Y

-Y (XLINK option). . . . 57, 67
-y (XLINK option) 57

Z

-Z (XLINK option) 9, 58
-z (XLINK option) 61
ZAX (linker output format). . . . 64
ZAX-I (linker output format) 67
ZAX-M (linker output format) 67
ZPAGE (segment type) 11

Symbols

__checksum (default label) 39
__checksum__value (default checksum value symbol). . . . 26
-A (XLINK option). . . . 31
-a (XLINK option) 31
-B (XLINK option). . . . 32, 79
-b (XLINK option) 9, 32
-C (XLINK option). . . . 34
-c (XLINK option) 34, 76
-D (XLINK option). . . . 34
-d (XLINK option) 34
-E (XLINK option) 34
-e (XLINK option) 35
-F (XLINK option) 35, 52, 77
-f (XLINK option) 35, 76
-G (XLINK option). . . . 36
-g (XLINK option) 36
-H (XLINK option). . . . 22, 36, 38
-h (XLINK option) 36
-I (XLINK option) 37
-J (XLINK option) 22, 38
-K (XLINK option). . . . 9, 41

-L (XLINK option) 42
-l (XLINK option). 42
-M (XLINK option) 9, 42
-N (XLINK option). 45
-n (XLINK option) 45
-o (XAR option) 114
-O (XLINK option). 46
-o (XLINK option) 47
-P (XLINK option) 9, 47
-p (XLINK option) 49, 77
-Q (XLINK option). 49
-q (XLINK option) 51
-R (XLINK option). 51
-r (XLINK option) 52
-rt (XLINK option) 52
-S (XLINK option) 52
-s (XLINK option) 52
-U (XLINK option). 53
-V (XAR option). 114
-V (XLINK option). 54
-w (XLINK option). 55
-x (XLINK option) 15–19, 56
-Y (XLINK option). 57, 67
-y (XLINK option) 57
-Z (XLINK option) 9, 58
-z (XLINK option) 61
--image_input (XLINK option). 38
--misrac (XLINK option) 44
--misrac_verbose (XLINK option) 45
#define (XLINK option) 34