

IAR C/C++ Development Guide

Compiling and Linking

for Advanced RISC Machines Ltd's
ARM[®] Cores



COPYRIGHT NOTICE

© 1999–2011 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, The Code to Success, IAR KickStart Kit, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB. J-Link and J-Trace are trademarks licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

ARM, Thumb, and Cortex are registered trademarks of Advanced RISC Machines Ltd.

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

Eighth edition: October 2011

Part number: DARM-8

This guide applies to version 6.3x of IAR Embedded Workbench® for ARM®.

Internal reference: M11, Too6.3, csrcarm6.30, csrct2011.2, V_111012, ISUD.

Brief contents

Tables	19
Preface	21
Part 1. Using the build tools	29
Introduction to the IAR build tools	31
Developing embedded applications	37
Data storage	53
Functions	57
Linking using ILINK	67
Linking your application	81
The DLIB runtime environment	93
Assembler language interface	129
Using C	151
Using C++	161
Application-related considerations	173
Efficient coding for embedded applications	187
Part 2. Reference information	207
External interface details	209
Compiler options	219
Linker options	261
Data representation	287
Extended keywords	303

Pragma directives	317
Intrinsic functions	337
The preprocessor	367
Library functions	373
The linker configuration file	383
Section reference	405
Stack usage control files	411
IAR utilities	417
Implementation-defined behavior for Standard C	445
Implementation-defined behavior for C89	461
Index	473

Contents

Tables	19
Preface	21
Who should read this guide	21
How to use this guide	21
What this guide contains	22
Other documentation	23
User and reference guides	24
The online help system	24
Further reading	24
Web sites	25
Document conventions	26
Typographic conventions	26
Naming conventions	27
 Part I. Using the build tools	29
Introduction to the IAR build tools	31
The IAR build tools—an overview	31
IAR C/C++ Compiler	31
IAR Assembler	32
The IAR ILINK Linker	32
Specific ELF tools	32
External tools	32
IAR language overview	33
Device support	33
Supported ARM devices	34
Preconfigured support files	34
Examples for getting started	34
Special support for embedded systems	34
Extended keywords	35
Pragma directives	35

Predefined symbols	35
Special function types	35
Accessing low-level features	35
Developing embedded applications	37
Developing embedded software using IAR build tools	37
Mapping of internal and external memory	37
Communication with peripheral units	38
Event handling	38
System startup	38
Real-time operating systems	38
Interoperability with other build tools	39
The build process—an overview	39
The translation process	40
The linking process	40
After linking	42
Application execution—an overview	42
The initialization phase	43
The execution phase	47
The termination phase	47
Building applications—an overview	47
Basic project configuration	48
Processor configuration	48
Optimization for speed and size	49
Runtime environment	50
Data storage	53
Introduction	53
Different ways to store data	53
Auto variables—on the stack	54
The stack	54
Dynamic memory on the heap	55
Functions	57
Function-related extensions	57

ARM and Thumb code	57
Execution in RAM	58
Primitives for interrupts, concurrency, and OS-related programming	59
Interrupt functions	59
Installing exception functions	60
Interrupts and fast interrupts	61
Nested interrupts	61
Software interrupts	62
Interrupt operations	63
Interrupts for ARM Cortex-M	64
C++ and special function types	65
Linking using ILINK	67
Linking—an overview	67
Modules and sections	68
The linking process	69
Placing code and data—the linker configuration file	71
A simple example of a configuration file	71
Initialization at system startup	74
The initialization process	74
C++ dynamic initialization	76
Stack usage analysis	76
Situations where warnings are issued	77
Limitations	77
Stack usage control files	78
Source annotation	78
Map file contents	78
Call graph output	79
Linking your application	81
Linking considerations	81
Choosing a linker configuration file	81
Defining your own memory areas	82
Placing sections	83

Reserving space in RAM	84
Keeping modules	85
Keeping symbols and sections	85
Application startup	85
Setting up the stack	85
Setting up the heap	86
Setting up the atexit limit	86
Changing the default initialization	86
Interaction between ILINK and the application	89
Standard library handling	89
Producing other output formats than ELF/DWARF	90
Veneers	90
Hints for troubleshooting	90
Relocation errors	90
The DLIB runtime environment	93
Introduction to the runtime environment	93
Runtime environment functionality	93
Setting up the runtime environment	94
Using prebuilt libraries	95
Library filename syntax	96
Groups of library files	97
Customizing a prebuilt library without rebuilding	98
Choosing formatters for printf and scanf	99
Choosing a printf formatter	99
Choosing a scanf formatter	100
Application debug support	101
Including C-SPY debugging support	101
The debug library functionality	102
The C-SPY Terminal I/O window	103
Low-level functions in the debug library	103
Adapting the library for target hardware	104
Library low-level interface	104
Overriding library modules	105

Building and using a customized library	105
Setting up a library project	106
Modifying the library functionality	106
Using a customized library	107
System startup and termination	107
System startup	107
System termination	110
Customizing system initialization	111
__low_level_init	111
Modifying the file cstartup.s	112
Library configurations	112
Choosing a runtime configuration	113
Standard streams for input and output	113
Implementing low-level character input and output	113
Configuration symbols for printf and scanf	115
Customizing formatting capabilities	116
File input and output	117
Locale	117
Locale support in prebuilt libraries	118
Customizing the locale support	118
Changing locales at runtime	119
Environment interaction	120
The getenv function	120
The system function	120
Signal and raise	121
Time	121
Pow	121
Assert	122
Atexit	122
Managing a multithreaded environment	122
Multithread support in the DLIB library	123
Enabling multithread support	123
Changes in the linker configuration file	127

Checking module consistency	127
Runtime model attributes	127
Using runtime model attributes	128
Assembler language interface	129
Mixing C and assembler	129
Intrinsic functions	129
Mixing C and assembler modules	130
Inline assembler	131
This is an example of how to use clobbered memory:	136
Calling assembler routines from C	136
Creating skeleton code	137
Compiling the code	137
Calling assembler routines from C++	138
Calling convention	139
Function declarations	140
Using C linkage in C++ source code	140
Preserved versus scratch registers	141
Function entrance	141
Function exit	143
Examples	144
Call frame information	146
CFI directives	146
Creating assembler source with CFI support	147
Using C	151
C language overview	151
Extensions overview	152
Enabling language extensions	153
IAR C language extensions	153
Extensions for embedded systems programming	154
Relaxations to Standard C	156

Using C++	161
Overview—Embedded C++ and Extended EC++	161
Embedded C++	161
Extended Embedded C++	162
Overview—Standard C++	162
Modes for exceptions and RTTI support	163
Exception handling	164
Enabling support for C++ and variants	165
C++ and EC++ feature descriptions	166
Using IAR attributes with Classes	166
Function types	167
Using static class objects in interrupts	167
Using New handlers	167
Templates	168
Debug support in C-SPY	168
EEC++ feature description	169
Templates	169
Variants of cast operators	169
Mutable	169
Namespace	169
The STD namespace	169
EC++ and C++ language extensions	170
Application-related considerations	173
Output format considerations	173
Stack considerations	173
Stack size considerations	174
Stack alignment	174
Exception stacks	174
Heap considerations	175
Interaction between the tools and your application	176
Checksum calculation	177
Calculating a checksum	178
Adding a checksum function to your source code	179

Things to remember	181
C-SPY considerations	181
Linker optimizations	181
Virtual Function Elimination	181
AEABI compliance	182
Linking AEABI-compliant modules using the IAR ILINK Linker ..	183
Linking AEABI-compliant modules using a third-party linker	183
Enabling AEABI compliance in the compiler	183
CMSIS integration	184
CMSIS DSP library	184
Customizing the CMSIS DSP library	184
Building with CMSIS on the command line	185
Building with CMSIS in IAR Embedded Workbench	185
Efficient coding for embedded applications	187
Selecting data types	187
Using efficient data types	187
Floating-point types	188
Alignment of elements in a structure	189
Anonymous structs and unions	189
Controlling data and function placement in memory	191
Data placement at an absolute location	192
Data and function placement in sections	193
Data placement in registers	194
Controlling compiler optimizations	195
Scope for performed optimizations	195
Multi-file compilation units	196
Optimization levels	196
Speed versus size	197
Fine-tuning enabled transformations	197
Facilitating good code generation	200
Writing optimization-friendly source code	200
Saving stack space and RAM memory	201
Function prototypes	201

Integer types and bit negation	202
Protecting simultaneously accessed variables	203
Accessing special function registers	203
Passing values between C and assembler objects	204
Non-initialized variables	205
Part 2. Reference information	207
External interface details	209
Invocation syntax	209
Compiler invocation syntax	209
ILINK invocation syntax	209
Passing options	210
Environment variables	211
Include file search procedure	211
Compiler output	212
ILINK output	214
Diagnostics	214
Message format for the compiler	214
Message format for the linker	215
Severity levels	215
Setting the severity level	216
Internal error	216
Compiler options	219
Options syntax	219
Types of options	219
Rules for specifying parameters	219
Summary of compiler options	222
Descriptions of compiler options	225

Linker options	261
Summary of linker options	261
Descriptions of linker options	263
Data representation	287
Alignment	287
Alignment on the ARM core	287
Byte order	288
Basic data types	288
Integer types	288
Floating-point types	294
Pointer types	296
Function pointers	296
Data pointers	296
Casting	296
Structure types	297
Alignment	297
General layout	297
Packed structure types	298
Type qualifiers	299
Declaring objects volatile	299
Declaring objects volatile and const	300
Declaring objects const	301
Data types in C++	301
Extended keywords	303
General syntax rules for extended keywords	303
Type attributes	303
Object attributes	305
Summary of extended keywords	306
Descriptions of extended keywords	307

Pragma directives	317
Summary of pragma directives	317
Descriptions of pragma directives	319
Intrinsic functions	337
Summary of intrinsic functions	337
Intrinsic functions for Neon instructions	345
Descriptions of intrinsic functions	346
The preprocessor	367
Overview of the preprocessor	367
Descriptions of predefined preprocessor symbols	368
Descriptions of miscellaneous preprocessor extensions	371
Library functions	373
Library overview	373
Header files	373
Library object files	374
Alternative more accurate library functions	374
Reentrancy	374
The longjmp function	375
IAR DLIB Library	375
C header files	375
C++ header files	376
Library functions as intrinsic functions	379
Added C functionality	379
Symbols used internally by the library	381
The linker configuration file	383
Overview	383
Defining memories and regions	384
Regions	385
Section handling	388
Section selection	396
Using symbols, expressions, and numbers	400

Structural configuration	402
Section reference	405
Summary of sections	405
Descriptions of sections and blocks	406
Stack usage control files	411
Overview	411
Stack usage control directives	411
Syntactic components	413
IAR utilities	417
The IAR Archive Tool—iarchive	417
Invocation syntax	417
Summary of iarchive commands	418
Summary of iarchive options	419
Diagnostic messages	419
The IAR ELF Tool—ielftool	420
Invocation syntax	421
Summary of ielftool options	421
The IAR ELF Dumper for ARM—ielfdumparm	422
Invocation syntax	422
Summary of ielfdumparm options	423
The IAR ELF Object Tool—iobjmanip	423
Invocation syntax	423
Summary of iobjmanip options	424
Diagnostic messages	424
The IAR Absolute Symbol Exporter—ismexport	426
Invocation syntax	426
Summary of ismexport options	427
Steering files	427
Diagnostic messages	429
Descriptions of options	431

Implementation-defined behavior for Standard C 445

Descriptions of implementation-defined behavior 445

Implementation-defined behavior for C89 461

Descriptions of implementation-defined behavior 461

Index 473

Tables

1: Typographic conventions used in this guide	26
2: Naming conventions used in this guide	27
3: Sections holding initialized data	74
4: Description of a relocation error	91
5: Customizable items	98
6: Formatters for printf	99
7: Formatters for scanf	100
8: Functions with special meanings when linked with debug library	103
9: Library configurations	112
10: Descriptions of printf configuration symbols	116
11: Descriptions of scanf configuration symbols	116
12: Low-level I/O files	117
13: Library objects using TLS	123
14: Macros for implementing TLS allocation	125
15: Example of runtime model attributes	128
16: Inline assembler operand constraints	134
17: Supported constraint modifiers	134
18: List of valid clobbers	136
19: Registers used for passing parameters	142
20: Registers used for returning values	143
21: Call frame information resources defined in a names block	146
22: Language extensions	153
23: Section operators and their symbols	155
24: Exception stacks	174
25: Compiler optimization levels	196
26: Compiler environment variables	211
27: ILINK environment variables	211
28: Error return codes	213
29: Compiler options summary	222
30: Linker options summary	261
31: Integer types	288

32: Floating-point types	294
33: Extended keywords summary	306
34: Pragma directives summary	317
35: Intrinsic functions summary	337
36: Predefined symbols	368
37: Traditional Standard C header files—DLIB	375
38: C++ header files	377
39: <Standard template library header files	377
40: New Standard C header files—DLIB	378
41: Examples of section selector specifications	398
42: Section summary	405
43: iarchive parameters	418
44: iarchive commands summary	418
45: iarchive options summary	419
46: ielftool parameters	421
47: ielftool options summary	421
48: ielfdumparm parameters	422
49: ielfdumparm options summary	423
50: iobjmanip parameters	423
51: iobjmanip options summary	424
52: ielftool parameters	426
53: isymexport options summary	427
54: Message returned by strerror()—IAR DLIB library	460
55: Message returned by strerror()—IAR DLIB library	471

Preface

Welcome to the IAR C/C++ Development Guide for ARM®. The purpose of this guide is to provide you with detailed reference information that can help you to use the build tools to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

Who should read this guide

Read this guide if you plan to develop an application using the C or C++ language for the ARM core and need detailed reference information on how to use the build tools. You should have working knowledge of:

- The architecture and instruction set of the ARM core. Refer to the documentation from Advanced RISC Machines Ltd for information about the ARM core
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you start using the IAR C/C++ compiler and linker for ARM, you should read *Part 1. Using the build tools* in this guide.

When you are familiar with the compiler and linker and have already configured your project, you can focus more on *Part 2. Reference information*.

If you are new to using the IAR Systems build tools, we recommend that you first study the *IDE Project Management and Building Guide for ARM®*. This guide contains a product overview, tutorials that can help you get started, conceptual and user information about the IDE and the IAR C-SPY® Debugger, and corresponding reference information.

What this guide contains

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

Part 1. Using the build tools

- *Introduction to the IAR build tools* gives an introduction to the IAR build tools, which includes an overview of the tools, the programming languages, the available device support, and extensions provided for supporting specific features of the ARM core.
- *Developing embedded applications* gives the information you need to get started developing your embedded software using the IAR build tools.
- *Data storage* describes how to store data in memory.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Linking using ILINK* describes the linking process using the IAR ILINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using ILINK options and tailoring the linker configuration file.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Application-related considerations* discusses a selected range of application issues related to using the compiler and linker.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Reference information

- *External interface details* provides reference information about how the compiler and linker interact with their environment—the invocation syntax, methods for passing options to the compiler and linker, environment variables, the include file

search procedure, and the different types of compiler and linker output. The chapter also describes how the diagnostic system works.

- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Linker options* gives a summary of the options, and contains detailed reference information for each linker option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Extended keywords* gives reference information about each of the ARM-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about functions to use for accessing ARM-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *The linker configuration file* describes the purpose of the linker configuration file and describes its contents.
- *Section reference* gives reference information about the use of sections.
- *Stack usage control files* describes the syntax and semantics of stack usage control files.
- *IAR utilities* describes the IAR utilities that handle the ELF and DWARF object formats.
- *Implementation-defined behavior for Standard C* describes how the compiler handles the implementation-defined areas of Standard C.
- *Implementation-defined behavior for C89* describes how the compiler handles the implementation-defined areas of the C language standard C89.

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

The complete set of IAR Systems development tools is described in a series of guides. For information about:

- System requirements and information about how to install and register the IAR Systems products, refer to the booklet *Quick Reference* (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, see the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, see the *IDE Project Management and Building Guide for ARM®*.
- Using the IAR C-SPY® Debugger, see the *C-SPY® Debugging Guide for ARM®*.
- Programming for the IAR Assembler for ARM, see the *ARM® IAR Assembler Reference Guide*.
- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for ARM, see the *IAR Embedded Workbench® Migration Guide for ARM®*.
- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

Note: Additional documentation might be available depending on your product installation.

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Comprehensive information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

FURTHER READING

These books might be of interest to you when using the IAR Systems development tools:

- Seal, David, and David Jagger. *ARM Architecture Reference Manual*. Addison-Wesley

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Furber, Steve, *ARM System-on-Chip Architecture*. Addison-Wesley.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.
- Meyers, Scott. *Effective STL*. Addison-Wesley.
- Sloss, Andrew N. et al, *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++*. Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.

WEB SITES

Recommended web sites:

- The Advanced RISC Machines Ltd web site, **www.arm.com**, that contains information and news about the ARM cores. This site also contains information about the ARM Embedded Application Binary Interface (AEABI) for the ARM architecture, and directs you to documentation about the ELF/DWARF standards.
- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.
- The web site of the C standardization working group, **www.open-std.org/jtc1/sc22/wg14**.
- The web site of the C++ Standards Committee, **www.open-std.org/jtc1/sc22/wg21**.

- Finally, the Embedded C++ Technical Committee web site, www.caravan.net/ec2plus, that contains information about the Embedded C++ standard.

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 `arm\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.n\arm\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:


Style	Used for
computer	<ul style="list-style-type: none">• Source code examples and file paths.• Text on the command line.• Binary, hexadecimal, and octal numbers.
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a linker directive, inline assembler statement, or a stack usage control directive, [and] are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
{option}	A mandatory part of a linker directive, { and } are not part of the actual directive, but any [,], {, or } are part of the directive syntax
[option]	An optional part of a command.
a b c	An optional part of a command with alternatives.
{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.
<i>italic</i>	<ul style="list-style-type: none">• 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.

Table 1: Typographic conventions used in this guide




Style	Used for
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

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

NAMING CONVENTIONS

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

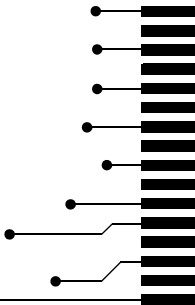
Brand name	Generic term
IAR Embedded Workbench® for ARM	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for ARM	the IDE
IAR C-SPY® Debugger for ARM	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for ARM	the compiler
IAR Assembler™ for ARM	the assembler
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide

Part I. Using the build tools

This part of the *IAR C/C++ Development Guide for ARM®* includes these chapters:

- Introduction to the IAR build tools
- Developing embedded applications
- Data storage
- Functions
- Linking using ILINK
- Linking your application
- The DLIB runtime environment
- Assembler language interface
- Using C
- Using C++
- Application-related considerations
- Efficient coding for embedded applications.





Introduction to the IAR build tools

This chapter gives an introduction to the IAR build tools for the ARM core, which means you will get an overview of:

- The IAR build tools—the build interfaces, compiler, assembler, and linker
- The programming languages
- The available device support
- The extensions provided by the IAR C/C++ Compiler for ARM to support specific features of the ARM core.

The IAR build tools—an overview

In the IAR product installation you can find a set of tools, code examples, and user documentation, all suitable for developing software for ARM-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.



IAR Embedded Workbench® is a very powerful Integrated Development Environment (IDE) that allows you to develop and manage complete embedded application projects. It provides an easy-to-learn and highly efficient development environment with maximum code inheritance capabilities, comprehensive and specific target support. IAR Embedded Workbench promotes a useful working methodology, and thus a significant reduction of the development time.

For information about the IDE, see the *IDE Project Management and Building Guide for ARM®*.



The compiler, assembler, and linker can also be run from a command line environment, if you want to use them as external tools in an already established project environment.

IAR C/C++ COMPILER

The IAR C/C++ Compiler for ARM is a state-of-the-art compiler that offers the standard features of the C and C++ languages, plus extensions designed to take advantage of the ARM-specific facilities.

IAR ASSEMBLER

The IAR Assembler for ARM is a powerful relocating macro assembler with a versatile set of directives and expression operators. The assembler features a built-in C language preprocessor and supports conditional assembly.

The IAR Assembler for ARM uses the same mnemonics and operand syntax as the Advanced RISC Machines Ltd ARM Assembler, which simplifies the migration of existing code. For more information, see the *ARM® IAR Assembler Reference Guide*.

THE IAR ILINK LINKER

The IAR ILINK Linker for ARM is a powerful, flexible software tool for use in the development of embedded controller applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable input, multi-module, C/C++, or mixed C/C++ and assembler programs.

SPECIFIC ELF TOOLS

ILINK both uses and produces industry-standard ELF and DWARF as object format, additional IAR utilities that handle these formats are provided:

- The IAR Archive Tool—`iarchive`—creates and manipulates a library (archive) of several ELF object files
- The IAR ELF Tool—`ielftool`—performs various transformations on an ELF executable image (such as, fill, checksum, format conversion etc)
- The IAR ELF Dumper for ARM—`ielfdumparm`—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—`iobjmanip`—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—`ismexport`—exports absolute symbols from a ROM image file, so that they can be used when linking an add-on application.

Note: These ELF utilities are well-suited for object files produced by the tools from IAR Systems. Thus, we recommend using them instead of the GNU binary utilities.

EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IDE Project Management and Building Guide for ARM®*.

IAR language overview

There are two high-level programming languages you can use with the IAR C/C++ Compiler for ARM:

- C, the most widely used high-level programming language in the embedded systems industry. You can build freestanding applications that follow these standards:
 - Standard C—also known as C99. Hereafter, this standard is referred to as *Standard C* in this guide.
 - C89—also known as C94, C90, C89, and ANSI C. This standard is required when MISRA C is enabled.
- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. Any of these standards can be used:
 - Standard C++—can be used with different levels of support for exceptions and runtime type information (RTTI).
 - Embedded C++ (EC++)—a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
 - IAR Extended Embedded C++ (EEC++)—EC++ with additional features such as full template support, multiple inheritance, namespace support, the new cast operators, as well as the Standard Template Library (STL).

Each of the supported languages can be used in *strict* or *relaxed* mode, or relaxed with IAR extensions enabled. The strict mode adheres to the standard, whereas the relaxed mode allows some common deviations from the standard.

For more information about C, see the chapter *Using C*.

For more information about C++, Embedded C++, and Extended Embedded C++, see the chapter *Using C++*.

For information about how the compiler handles the implementation-defined areas of the languages, see the chapter *Implementation-defined behavior for Standard C*.

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *ARM® IAR Assembler Reference Guide*.

Device support

To get a smooth start with your product development, the IAR product installation comes with a wide range of device-specific support.

SUPPORTED ARM DEVICES

The IAR C/C++ Compiler for ARM supports several different ARM cores and devices based on the instruction sets version 4, 5, 6, 6M, and 7. The object code that the compiler generates is not always binary compatible between the cores. Therefore it is crucial to specify a processor option to the compiler. The default core is ARM7TDMI.

PRECONFIGURED SUPPORT FILES

The IAR product installation contains preconfigured files for supporting different devices. If you need additional files for device support, they can be created using one of the provided ones as a template.

Header files for I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. The product package supplies I/O files for all devices that are available at the time of the product release. You can find these files in the `arm\inc\<vendor>` directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can be created using one of the provided ones as a template. For detailed information about the header file format, see `EWARM_HeaderFormat.pdf` located in the `arm\doc\` directory.

Device description files

The debugger handles several of the device-specific requirements, such as definitions of peripheral registers and groups of these, by using device description files. These files are located in the `arm\inc` directory and they have the filename extension `ddf`. For more information about these files, see the *C-SPY® Debugging Guide for ARM®* and `EWARM_DDFFormat.pdf` located in the `arm\doc\` directory.

EXAMPLES FOR GETTING STARTED

The `arm\examples` directory contains several hundreds of examples of working applications to give you a smooth start with your development. The complexity of the examples ranges from simple LED blink to USB mass storage controllers. Examples are provided for most of the supported devices.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the ARM core.

EXTENDED KEYWORDS

The compiler provides a set of keywords that can be used for configuring how the code is generated. For example, there are keywords for declaring special function types.



By default, language extensions are enabled in the IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 235 for additional information.

For more information about the extended keywords, see the chapter *Extended keywords*.

PRAGMA DIRECTIVES

The pragma directives control the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it issues warning messages.

The pragma directives are always enabled in the compiler. They are consistent with standard C, and are very useful when you want to make sure that the source code is portable.

For more information about the pragma directives, see the chapter *Pragma directives*.

PREDEFINED SYMBOLS

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example the CPU mode and time of compilation.

For more information about the predefined symbols, see the chapter *The preprocessor*.

SPECIAL FUNCTION TYPES

The special hardware features of the ARM core are supported by the compiler's special function types: software interrupts, interrupts, and fast interrupts. You can write a complete application without having to write any of these functions in assembler language.

For more information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 59.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 129.

Developing embedded applications

This chapter provides the information you need to get started developing your embedded software for the ARM core using the IAR build tools.

First, you will get an overview of the tasks related to embedded software development, followed by an overview of the build process, including the steps involved for compiling and linking an application.

Next, the program flow of an executing application is described.

Finally, you will get an overview of the basic settings needed for a project.

Developing embedded software using IAR build tools

Typically, embedded software written for a dedicated microcontroller is designed as an endless loop waiting for some external events to happen. The software is located in ROM and executes on reset. You must consider several hardware and software factors when you write this kind of software.

MAPPING OF INTERNAL AND EXTERNAL MEMORY

Embedded systems typically contain various types of memory, such as on-chip RAM, external DRAM or SRAM, ROM, EEPROM, or flash memory.

As an embedded software developer, you must understand the features of the different types of memory. For example, on-chip RAM is often faster than other types of memories, and variables that are accessed often would in time-critical applications benefit from being placed here. Conversely, some configuration data might be accessed seldom but must maintain their value after power off, so they should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. For more information, see *Controlling data and function placement in memory*, page 191. The linker places sections of code and data in memory according to the directives you specify in the linker configuration file, see *Placing code and data—the linker configuration file*, page 71.

COMMUNICATION WITH PERIPHERAL UNITS

If external devices are connected to the microcontroller, you might need to initialize and control the signalling interface, for example by using chip select pins, and detect and handle external interrupt signals. Typically, this must be initialized and controlled at runtime. The normal way to do this is to use special function registers (SFR). These are typically available at dedicated addresses, containing bits that control the chip configuration.

Standard peripheral units are defined in device-specific I/O header files with the filename extension `.h`. See *Device support*, page 33. For an example, see *Accessing special function registers*, page 203.

EVENT HANDLING

In embedded systems, using *interrupts* is a method for handling external events immediately; for example, detecting that a button was pressed. In general, when an interrupt occurs in the code, the core immediately stops executing the code it runs, and starts executing an interrupt routine instead.

The compiler supports the following processor exception types: interrupts, software interrupts, and fast interrupts, which means that you can write your interrupt routines in C, see *Interrupt functions*, page 59.

SYSTEM STARTUP

In all embedded systems, system startup code is executed to initialize the system—both the hardware and the software system—before the `main` function of the application is called. The CPU imposes this by starting execution from a fixed memory address.

As an embedded software developer, you must ensure that the startup code is located at the dedicated memory addresses, or can be accessed using a pointer from the vector table. This means that startup code and the initial vector table must be placed in non-volatile memory, such as ROM, EPROM, or flash.

A C/C++ application further needs to initialize all global variables. This initialization is handled by the linker and the system startup code in conjunction. For more information, see *Application execution—an overview*, page 42.

REAL-TIME OPERATING SYSTEMS

In many cases, the embedded application is the only software running in the system. However, using an RTOS has some advantages.

For example, the timing of high-priority tasks is not affected by other parts of the program which are executed in lower priority tasks. This typically makes a program more deterministic and can reduce power consumption by using the CPU efficiently and putting the CPU in a lower-power state when idle.

Using an RTOS can make your program easier to read and maintain, and in many cases smaller as well. Application code can be cleanly separated in tasks which are truly independent of each other. This makes teamwork easier, as the development work can be easily split into separate tasks which are handled by one developer or a group of developers.

Finally, using an RTOS reduces the hardware dependence and creates a clean interface to the application, making it easier to port the program to different target hardware.

INTEROPERABILITY WITH OTHER BUILD TOOLS

The IAR compiler and linker provide support for AEABI, the Embedded Application Binary Interface for ARM. For more information about this interface specification, see the www.arm.com web site.

The advantage of this interface is the interoperability between vendors supporting it; an application can be built up of libraries of object files produced by different vendors and linked with a linker from any vendor, as long as they adhere to the AEABI standard.

AEABI specifies full compatibility for C and C++ object code, and for the C library. The AEABI does not include specifications for the C++ library.

For more information about the AEABI support in the IAR build tools, see *AEABI compliance*, page 182.

The ARM IAR build tools version 6.xx are not fully compatible with earlier versions of the product, see the *IAR Embedded Workbench® Migration Guide for ARM®* for more information.

For more information, see *Linker optimizations*, page 181.

The build process—an overview

This section gives an overview of the build process; how the various build tools—compiler, assembler, and linker—fit together, going from source code to an executable image.

To get familiar with the process in practice, you should run one or more of the tutorials available from the IAR Information Center.

THE TRANSLATION PROCESS

There are two tools in the IDE that translate application source files to intermediary object files. The IAR C/C++ Compiler and the IAR Assembler. Both produce relocatable object files in the industry-standard format ELF, including the DWARF format for debug information.

Note: The compiler can also be used for translating C/C++ source code into assembler source code. If required, you can modify the assembler source code which then can be assembled into object code. For more information about the IAR Assembler, see the *ARM® IAR Assembler Reference Guide*.

This illustration shows the translation process:

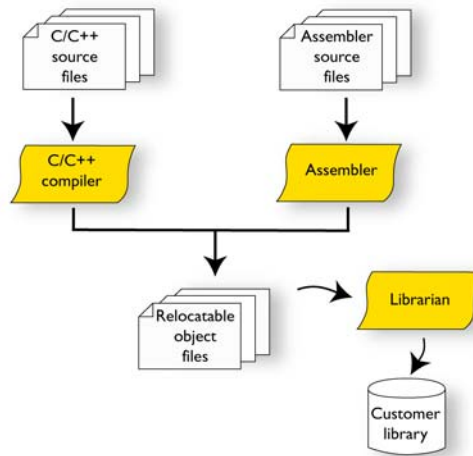


Figure 1: The build process before linking

After the translation, you can choose to pack any number of modules into an archive, or in other words, a library. The important reason you should use libraries is that each module in a library is conditionally linked in the application, or in other words, is only included in the application if the module is used directly or indirectly by a module supplied as an object file. Optionally, you can create a library; then use the IAR utility `iarchive`.

THE LINKING PROCESS

The relocatable modules, in object files and libraries, produced by the IAR compiler and assembler cannot be executed as is. To become an executable application, they must be *linked*.

Note: Modules produced by a toolset from another vendor can be included in the build as well. Be aware that this also might require a compiler utility library from the same vendor.

The IAR ILINK Linker (`ilinkarm.exe`) is used for building the final application. Normally, ILINK requires the following information as input:

- Several object files and possibly certain libraries
- A program start label (set by default)
- The linker configuration file that describes placement of code and data in the memory of the target system.

This illustration shows the linking process:

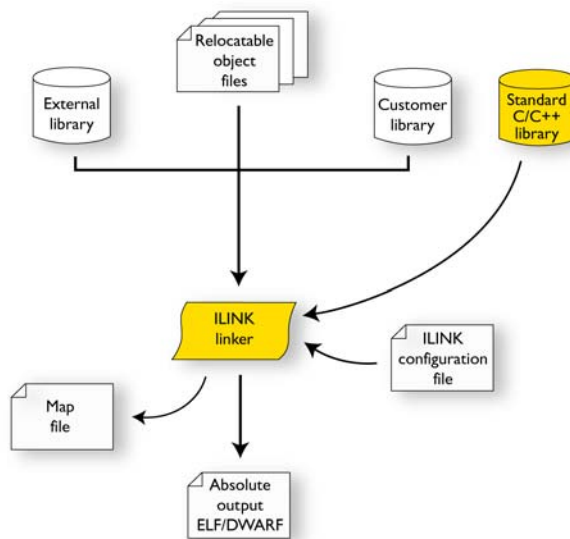


Figure 2: The linking process

Note: The Standard C/C++ library contains support routines for the compiler, and the implementation of the C/C++ standard library functions.

During the linking, ILINK might produce error messages and logging messages on `stdout` and `stderr`. The log messages are useful for understanding why an application was linked the way it was, for example, why a module was included or a section removed.

For more information about the procedure performed by ILINK, see *The linking process*, page 69.

AFTER LINKING

The IAR ILINK Linker produces an absolute object file in ELF format that contains the executable image. After linking, the produced absolute executable image can be used for:

- Loading into the IAR C-SPY Debugger or any other compatible external debugger that reads ELF and DWARF.
- Programming to a flash/PROM using a flash/PROM programmer. Before this is possible, the actual bytes in the image must be converted into the standard Motorola 32-bit S-record format or the Intel Hex-32 format. For this, use `ielftool`, see *The IAR ELF Tool—ielftool*, page 420.

This illustration shows the possible uses of the absolute output ELF/DWARF file:

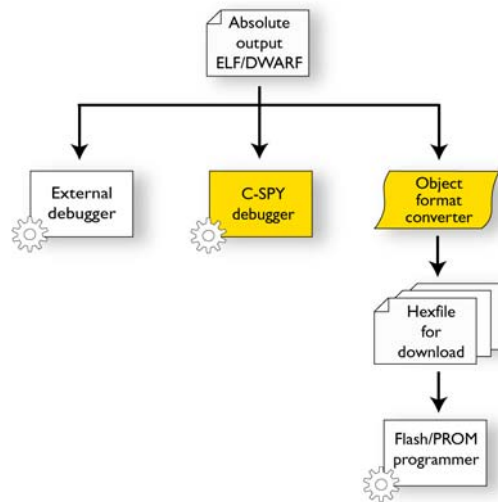


Figure 3: Possible uses of the absolute output ELF/DWARF file

Application execution—an overview

This section gives an overview of the execution of an embedded application divided into three phases, the:

- Initialization phase

- Execution phase
- Termination phase.

THE INITIALIZATION PHASE

Initialization is executed when an application is started (the CPU is reset) but before the `main` function is entered. The initialization phase can for simplicity be divided into:

- Hardware initialization, which generally at least initializes the stack pointer.
The hardware initialization is typically performed in the system startup code `cstartup.s` and if required, by an extra low-level routine that you provide. It might include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.
- Software C/C++ system initialization
Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.
- Application initialization
This depends entirely on your application. It can include setting up an RTOS kernel and starting initial tasks for an RTOS-driven application. For a bare-bone application, it can include setting up various interrupts, initializing communication, initializing devices, etc.

For a ROM/flash-based system, constants and functions are already placed in ROM. All symbols placed in RAM must be initialized before the `main` function is called. The linker has already divided the available RAM into different areas for variables, stack, heap, etc.

The following sequence of illustrations gives a simplified overview of the different stages of the initialization.

- I When an application is started, the system startup code first performs hardware initialization, such as initialization of the stack pointer to point at the end of the predefined stack area:

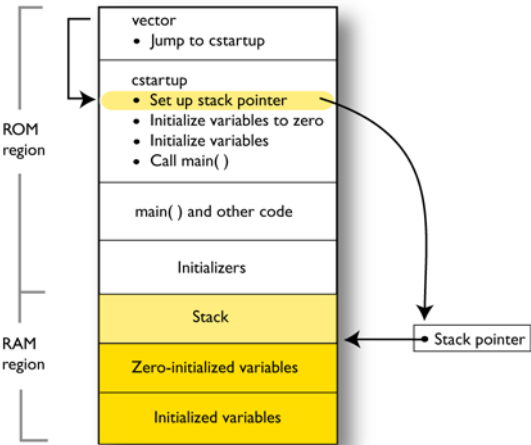


Figure 4: Initializing hardware

- 2 Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

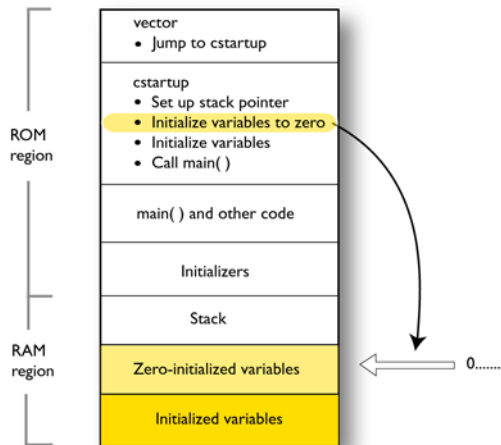


Figure 5: Zero-initializing variables

Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`

- 3 For *initialized data*, data declared, for example, like `int i = 6`; the initializers are copied from ROM to RAM:

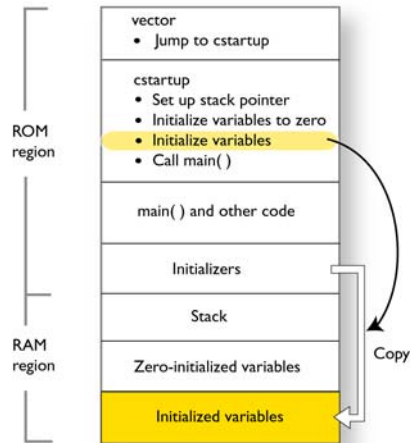


Figure 6: Initializing variables

- 4 Finally, the `main` function is called:

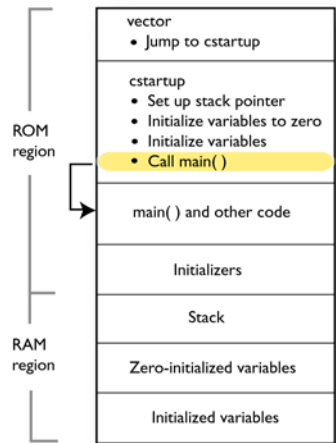


Figure 7: Calling main

For more information about each stage, see *System startup and termination*, page 107. For more information about initialization of data, see *Initialization at system startup*, page 74.

THE EXECUTION PHASE

The software of an embedded application is typically implemented as a loop which is either interrupt-driven or uses polling for controlling external interaction or internal events. For an interrupt-driven system, the interrupts are typically initialized at the beginning of the `main` function.

In a system with real-time behavior and where responsiveness is critical, a multi-task system might be required. This means that your application software should be complemented with a real-time operating system. In this case, the RTOS and the different tasks must also be initialized at the beginning of the `main` function.

THE TERMINATION PHASE

Typically, the execution of an embedded application should never end. If it does, you must define a proper end behavior.

To terminate an application in a controlled way, either call one of the Standard C library functions `exit`, `_Exit`, or `abort`, or return from `main`. If you return from `main`, the `exit` function is executed, which means that C++ destructors for static and global variables are called (C++ only) and all open files are closed.

Of course, in case of incorrect program logic, the application might terminate in an uncontrolled and abnormal way—a system crash.

For more information about this, see *System termination*, page 110.

Building applications—an overview

In the command line interface, this line compiles the source file `myfile.c` into the object file `myfile.o` using the default settings:

```
iccarm myfile.c
```

On the command line, this line can be used for starting ILINK:

```
ilinkarm myfile.o myfile2.o -o a.out --config my_configfile.icf
```

In this example, `myfile.o` and `myfile2.o` are object files, and `my_configfile.icf` is the linker configuration file. The option `-o` specifies the name of the output file.

Note: By default, the label where the application starts is `__iar_program_start`. You can use the `--entry` command line option to change this.

Basic project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler and linker generate the best code for the ARM device you are using. You can specify the options either from the command line interface or in the IDE.

You need to make settings for:

- Processor configuration, that is processor variant, CPU mode, interworking, VFP and floating-point arithmetic, and byte order
- Optimization settings
- Runtime environment
- Customizing the ILINK configuration, see the chapter *Linking your application*.

In addition to these settings, many other options and settings can fine-tune the result even further. For information about how to set options and for a list of all available options, see the chapters *Compiler options*, *Linker options*, and the *IDE Project Management and Building Guide for ARM®*, respectively.

PROCESSOR CONFIGURATION

To make the compiler generate optimum code, you should configure it for the ARM core you are using.

Processor variant

The IAR C/C++ Compiler for ARM supports several different ARM cores and devices based on the instruction sets version 4, 5, 6, and 7. All supported cores support Thumb instructions and 64-bit multiply instructions. The object code that the compiler generates is not always binary compatible between the cores. Therefore it is crucial to specify a processor option to the compiler. The default core is ARM7TDMI.



See the *IDE Project Management and Building Guide for ARM®* for information about setting the **Processor variant** option in the IDE.



Use the `--cpu` option to specify the ARM core; see *--cpu*, page 228 for syntax information.

CPU mode

The IAR C/C++ Compiler for ARM supports two CPU modes: ARM and Thumb.

All functions and function pointers will compile in the mode that you specify, except those explicitly declared `__arm` or `__thumb`.



See the *IDE Project Management and Building Guide for ARM®* for information about setting the **Processor variant** or **Chip** option in the IDE.



Use the `--arm` or `--thumb` option to specify the CPU mode for your project; see `--arm`, page 227 and `--thumb`, page 257, for syntax information.

Interworking

When code is compiled with the `--interwork` option, ARM and Thumb code can be freely mixed. Interworking functions can be called from both ARM and Thumb code. Interworking is default for devices based on the instruction sets version 5, 6, and 7, or when using the `--aeabi` compiler option.



See the *IDE Project Management and Building Guide for ARM®* for information about setting the **Generate interwork code** option in the IDE.



Use the `--interwork` option to specify interworking capabilities for your project; see `--interwork`, page 239, for syntax information.

VFP and floating-point arithmetic

If you are using an ARM core that contains a Vector Floating Point (VFP) coprocessor, you can use the `--fpu` option to generate code that carries out floating-point operations utilizing the coprocessor, instead of using the software floating-point library routines.



See the *IDE Project Management and Building Guide for ARM®* for information about setting the **FPU** option in the IDE.



Use the `--fpu` option to use the coprocessor for floating-point operations; see `--fpu`, page 238, for syntax information.

Byte order

The IAR C/EC++ Compiler for ARM supports the big-endian and little-endian byte order. All user and library modules in your application must use the same byte order.



See the *IDE Project Management and Building Guide for ARM®* for information about setting the **Endian mode** option in the IDE.



Use the `--endian` option to specify the byte order for your project; see `--endian`, page 236, for syntax information.

OPTIMIZATION FOR SPEED AND SIZE

The compiler's optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common subexpression elimination, static clustering, instruction scheduling, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

You can decide between several optimization levels and for the highest level you can choose between different optimization goals—*size*, *speed*, or *balanced*. Most optimizations will make the application both smaller and faster. However, when this is not the case, the compiler uses the selected optimization goal to decide how to perform the optimization.

The optimization level and goal can be specified for the entire application, for individual files, and for individual functions. In addition, some individual optimizations, such as function inlining, can be disabled.

For information about compiler optimizations and for more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

RUNTIME ENVIRONMENT

To create the required runtime environment you should choose a runtime library and set library options. You might also need to override certain library modules with your own customized versions.

The runtime library provided is the IAR DLIB Library, which supports Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.

The runtime library contains the functions defined by the C and the C++ standards, and include files that define the library interface (the system header files).

The runtime library you choose can be one of the prebuilt libraries, or a library that you customized and built yourself. The IDE provides a library project template, that you can use for building your own library version. This gives you full control of the runtime environment. If your project only contains assembler source code, you do not need to choose a runtime library.

For more information about the runtime environment, see the chapter *The DLIB runtime environment*.



Setting up for the runtime environment in the IDE

The library is automatically chosen according to the settings you make in **Project>Options>General Options**, on the pages **Target**, **Library Configuration**, **Library Options**. A correct include path is automatically set up for the system header files and for the device-specific include files.

Note that for the DLIB library there are different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, etc. See *Library configurations, page 112*, for more information.



Setting up for the runtime environment from the command line

You do not have to specify a library file explicitly, as ILINK automatically uses the correct library file.

A library configuration file that matches the library object file is automatically used. To explicitly specify a library configuration, use the `--dlib_config` option.

In addition to these options you might want to specify any target-specific linker options or the include path to application-specific header files by using the `-I` option, for example:

```
-I arm\inc
```

For information about the prebuilt library object files, see *Using prebuilt libraries*, page 95 (DLIB).

Setting library and runtime environment options

You can set certain options to reduce the library and runtime environment size:

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 99.
- The size of the stack and the heap, see *Setting up the stack*, page 85, and *Setting up the heap*, page 86, respectively.

Data storage

This chapter gives a brief introduction to the memory layout of the ARM core and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. Finally, detailed information about data storage on the stack and the heap is provided.

Introduction

An ARM core can address 4 Gbytes of continuous memory, ranging from 0x00000000 to 0xFFFFFFFF. Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

DIFFERENT WAYS TO STORE DATA

In a typical application, data can be stored in memory in three different ways:

- Auto variables

All variables that are local to a function, except those declared static, are stored on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid.

- Global variables, module-static variables, and local variables declared `static`

In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. The ARM core has one single address space and the compiler supports full memory addressing.

- Dynamically allocated data.

An application can allocate data on the *heap*, where the data remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 55.

Auto variables—on the stack

Variables that are defined inside a function—and not declared static—are named *auto variables* by the C standard. A few of these variables are placed in processor registers; the rest are placed on the stack. From a semantic point of view, this is equivalent. The main differences are that accessing registers is faster, and that less memory is required compared to when variables are located on the stack.

Auto variables can only live as long as the function executes; when the function returns, the memory allocated on the stack is released.

THE STACK

The stack can contain:

- Local variables and parameters not stored in registers
- Temporary results of expressions
- The return value of a function (unless it is passed in registers)
- Processor state during interrupts
- Processor registers that should be restored before the function returns (callee-save registers).

The stack is a fixed block of memory, divided into two parts. The first part contains allocated memory used by the function that called the current function, and the function that called it, etc. The second part contains free memory that can be allocated. The borderline between the two areas is called the *top of stack* and is represented by the stack pointer, which is a dedicated processor register. Memory is allocated on the stack by moving the stack pointer.

A function should never refer to the memory in the area of the stack that contains free memory. The reason is that if an interrupt occurs, the called interrupt function can allocate, modify, and—of course—deallocate memory on the stack.

Advantages

The main advantage of the stack is that functions in different parts of the program can use the same memory space to store their data. Unlike a heap, a stack will never become fragmented or suffer from memory leaks.

It is possible for a function to call itself either directly or indirectly—a *recursive function*—and each invocation can store its own data on the stack.

Potential problems

The way the stack works makes it impossible to store data that is supposed to live after the function returns. The following function demonstrates a common programming mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns.

```
int *MyFunction()
{
    int x;
    /* Do something here. */
    return &x; /* Incorrect */
}
```

Another problem is the risk of running out of stack. This will happen when one function calls another, which in turn calls a third, etc., and the sum of the stack usage of each function is larger than the size of the stack. The risk is higher if large data objects are stored on the stack, or when recursive functions are used.

Dynamic memory on the heap

Memory for objects allocated on the heap will live until the objects are explicitly released. This type of memory storage is very useful for applications where the amount of data is not known until runtime.

In C, memory is allocated using the standard library function `malloc`, or one of the related functions `calloc` and `realloc`. The memory is released again using `free`.

In C++, a special keyword, `new`, allocates memory and runs constructors. Memory allocated with `new` must be released using the keyword `delete`.

For related information, see *--basic_heap*, page 264.

Potential problems

Applications that are using heap-allocated objects must be designed very carefully, because it is easy to end up in a situation where it is not possible to allocate objects on the heap.

The heap can become exhausted if your application uses too much memory. It can also become full if memory that no longer is in use was not released.

For each allocated memory block, a few bytes of data for administrative purposes is required. For applications that allocate a large number of small blocks, this administrative overhead can be substantial.

There is also the matter of *fragmentation*; this means a heap where small sections of free memory is separated by memory used by allocated objects. It is not possible to allocate a new object if no piece of free memory is large enough for the object, even though the sum of the sizes of the free memory exceeds the size of the object.

Unfortunately, fragmentation tends to increase as memory is allocated and released. For this reason, applications that are designed to run for a long time should try to avoid using memory allocated on the heap.

Functions

This chapter contains information about functions. It gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.

Function-related extensions

In addition to supporting Standard C, the compiler provides several extensions for writing functions in C. Using these, you can:

- Generate code for the different CPU modes ARM and Thumb
- Use primitives for interrupts, concurrency, and OS-related programming
- Facilitate function optimization
- Access hardware features.

The compiler uses compiler options, extended keywords, pragma directives, and intrinsic functions to support this.

For more information about optimizations, see *Efficient coding for embedded applications*, page 187. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

ARM and Thumb code

The IAR C/C++ Compiler for ARM can generate code for either the 32-bit ARM, or the 16-bit Thumb or Thumb2 instruction set. Use the `--cpu_mode` option, alternatively the `--arm` or `--thumb` options, to specify which instruction set should be used for your project. For individual functions, it is possible to override the project setting by using the extended keywords `__arm` and `__thumb`. You can freely mix ARM and thumb code in the same application, as long as the code is interworking.

When performing function calls, the compiler always attempts to generate the most efficient assembler language instruction or instruction sequence available. As a result, 4 Gbytes of continuous memory in the range `0x0-0xFFFFFFFF` can be used for placing code. There is a limit of 4 Mbytes per code module.

The size of all code pointers is 4 bytes. There are restrictions to implicit and explicit casts from code pointers to data pointers or integer types or vice versa. For further information about the restrictions, see *Pointer types*, page 296.

In the chapter *Assembler language interface*, the generated code is studied in more detail in the description of calling C functions from assembler language and vice versa.

Execution in RAM

The `__ramfunc` keyword makes a function execute in RAM. In other words it places the function in a section that has read/write attributes. The function is copied from ROM to RAM at system startup just like any initialized variable, see *System startup and termination*, page 107.

The keyword is specified before the return type:

```
__ramfunc void foo(void);
```

If a function declared `__ramfunc` tries to access ROM, the compiler will issue a warning.

If the whole memory area used for code and constants is disabled—for example, when the whole flash memory is being erased—only functions and data stored in RAM may be used. Interrupts must be disabled unless the interrupt vector and the interrupt service routines are also stored in RAM.

String literals and other constants can be avoided by using initialized variables. For example, the following lines:

```
__ramfunc void test()
{
    /* myc: initializer in ROM */
    const int myc[] = { 10, 20 };

    /* string literal in ROM */
    msg("Hello");
}
```

can be rewritten to:

```
__ramfunc void test()
{
    /* myc: initialized by cstartup */
    static int myc[] = { 10, 20 };

    /* hello: initialized by cstartup */
    static char hello[] = "Hello";

    msg(hello);
}
```

For more details, see *Initializing code—copying ROM to RAM*, page 88.

Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for ARM provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__irq`, `__fiq`, `__swi`, and `__nested`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, and `__set_interrupt_state`.

Note: ARM Cortex-M has a different interrupt mechanism than other ARM devices, and for these devices a different set of primitives is available. For more details, see *Interrupts for ARM Cortex-M*, page 64.

INTERRUPT FUNCTIONS

In embedded systems, using interrupts is a method for handling external events immediately; for example, detecting that a button was pressed.

Interrupt service routines

In general, when an interrupt occurs in the code, the core immediately stops executing the code it runs, and starts executing an interrupt routine instead. It is important that the environment of the interrupted function is restored after the interrupt is handled (this includes the values of processor registers and the processor status register). This makes it possible to continue the execution of the original code after the code that handled the interrupt was executed.

The compiler supports interrupts, software interrupts, and fast interrupts. For each interrupt type, an interrupt routine can be written.

All interrupt functions must be compiled in ARM mode; if you are using Thumb mode, use the `__arm` extended keyword or the `#pragma type_attribute=__arm` directive to override the default behavior. This is not applicable for Cortex-M devices.

Interrupt vectors and the interrupt vector table

Each interrupt routine is associated with a vector address/instruction in the exception vector table, which is specified in the ARM cores documentation. The interrupt vector is the address in the exception vector table. For the ARM cores, the exception vector table starts at address `0x0`.

Defining an interrupt function—an example

To define an interrupt function, the `__irq` or the `__fiq` keyword can be used. For example:

```
__irq __arm void IRQ_Handler(void)
{
    /* Do something */
}
```

See the ARM cores documentation for more information about the interrupt vector table.

Note: An interrupt function must have the return type `void`, and it cannot specify any parameters.

INSTALLING EXCEPTION FUNCTIONS

All interrupt functions and software interrupt handlers must be installed in the vector table. This is done in assembler language in the system startup file `cstartup.s`.

The default implementation of the ARM exception vector table in the standard runtime library jumps to predefined functions that implement an infinite loop. Any exception that occurs for an event not handled by your application will therefore be caught in the infinite loop (B.).

The predefined functions are defined as weak symbols. A weak symbol is only included by the linker as long as no duplicate symbol is found. If another symbol is defined with the same name, it will take precedence. Your application can therefore simply define its own exception function by just defining it using the correct name.

These exception function names are defined in `cstartup.s` and referred to by the library exception vector code:

```
Undefined_Handler
SWI_Handler
Prefetch_Handler
Abort_Handler
IRQ_Handler
FIQ_Handler
```

To implement your own exception handler, define a function using the appropriate exception function name from the list above.

For example to add an interrupt function in C, it is sufficient to define an interrupt function named `IRQ_Handler`:

```
__irq __arm void IRQ_Handler()
{
}
```

An interrupt function must have C linkage, read more in *Calling convention*, page 139.

If you use C++, an interrupt function could look, for example, like this:

```
extern "C"
{
    __irq __arm void IRQ_Handler(void);
}

__irq __arm void IRQ_Handler(void)
{
}
```

No other changes are needed.

INTERRUPTS AND FAST INTERRUPTS

The interrupt and fast interrupt functions are easy to handle as they do not accept parameters or have a return value.

- To declare an interrupt function, use the `__irq` extended keyword or the `#pragma type_attribute=__irq` directive. For syntax information, see `__irq`, page 308, and `type_attribute`, page 333, respectively.
- To declare a fast interrupt function, use the `__fiq` extended keyword or the `#pragma type_attribute=__fiq` directive. For syntax information, see `__fiq`, page 308, and `type_attribute`, page 333, respectively.

Note: An interrupt function (`irq`) and a fast interrupt function (`fiq`) must have a return type of `void` and cannot have any parameters. A software interrupt function (`swi`) may have parameters and return values. By default, only four registers, `R0–R3`, can be used for parameters and only the registers `R0–R1` can be used for return values.

NESTED INTERRUPTS

Interrupts are automatically disabled by the ARM core prior to entering an interrupt handler. If an interrupt handler re-enables interrupts, calls functions, and another interrupt occurs, then the return address of the interrupted function—stored in `LR`—is overwritten when the second IRQ is taken. In addition, the contents of `SPSR` will be destroyed when the second interrupt occurs. The `__irq` keyword itself does not save and restore `LR` and `SPSR`. To make an interrupt handler perform the necessary steps needed when handling nested interrupts, the keyword `__nested` must be used in addition to `__irq`. The function prolog—function entrance sequence—that the compiler generates for nested interrupt handlers will switch from IRQ mode to system mode. Make sure that both the IRQ stack and system stack is set up. If you use the default `cstartup.s` file, both stacks are correctly set up.

Compiler-generated interrupt handlers that allow nested interrupts are supported for IRQ interrupts only. The FIQ interrupts are designed to be serviced quickly, which in most cases mean that the overhead of nested interrupts would be too high.

This example shows how to use nested interrupts with the ARM vectored interrupt controller (VIC):

```
__irq __nested __arm void interrupt_handler(void)
{
    void (*interrupt_task)();
    unsigned int vector;

    /* Get interrupt vector. */
    vector = VICVectAddr;

    /* Acknowledge interrupt in VIC. */
    VICVectAddr = 0;

    interrupt_task = (void(*)()) vector;

    /* Allow other IRQ interrupts to be serviced. */
    __enable_interrupt();

    /* Execute the task associated with this interrupt. */
    (*interrupt_task)();
}
```

Note: The `__nested` keyword requires the processor mode to be in either User or System mode.

SOFTWARE INTERRUPTS

Software interrupt functions are slightly more complex than other interrupt functions, in the way that they need a software interrupt handler (a dispatcher), are invoked (called) from running application software, and that they accept arguments and have return values. The mechanisms for calling a software interrupt function and how the software interrupt handler dispatches the call to the actual software interrupt function is described here.

Calling a software interrupt function

To call a software interrupt function from your application source code, the assembler instruction `SVC #immed` is used, where `immed` is an integer value that is referred to as the software interrupt number—or `swi_number`—in this guide. The compiler provides an easy way to implicitly generate this instruction from C/C++ source code, by using the `__swi` keyword and the `#pragma swi_number` directive when declaring the function.

A `__swi` function can for example be declared like this:

```
#pragma swi_number=0x23
__swi int swi_function(int a, int b);
```

In this case, the assembler instruction `SVC 0x23` will be generated where the function is called.

Software interrupt functions follow the same calling convention regarding parameters and return values as an ordinary function, except for the stack usage, see *Calling convention*, page 139.

For more information, see `__swi`, page 312, and `swi_number`, page 333, respectively.

The software interrupt handler and functions

The interrupt handler, for example `SWI_Handler` works as a dispatcher for software interrupt functions. It is invoked from the interrupt vector and is responsible for retrieving the software interrupt number and then calling the proper software interrupt function. The `SWI_Handler` must be written in assembler as there is no way to retrieve the software interrupt number from C/C++ source code.

The software interrupt functions

The software interrupt functions can be written in C or C++. Use the `__swi` keyword in a function definition to make the compiler generate a return sequence suited for a specific software interrupt function. The `#pragma swi_number` directive is not needed in the interrupt function definition.

For more information, see `__swi`, page 312.

Setting up the software interrupt stack pointer

If software interrupts will be used in your application, then the software interrupt stack pointer (`SVC_STACK`) must be set up and some space must be allocated for the stack. The `SVC_STACK` pointer can be set up together with the other stacks in the `cstartup.s` file. As an example, see the set up of the interrupt stack pointer. Relevant space for the `SVC_STACK` pointer is set up in the linker configuration file, see *Setting up the stack*, page 85.

INTERRUPT OPERATIONS

An interrupt function is called when an external event occurs. Normally it is called immediately while another function is executing. When the interrupt function has finished executing, it returns to the original function. It is imperative that the environment of the interrupted function is restored; this includes the value of processor registers and the processor status register.

When an interrupt occurs, the following actions are performed:

- The operating mode is changed corresponding to the particular exception
- The address of the instruction following the exception entry instruction is saved in R14 of the new mode
- The old value of the CPSR is saved in the SPSR of the new mode
- Interrupt requests are disabled by setting bit 7 of the CPSR and, if the exception is a fast interrupt, further fast interrupts are disabled by setting bit 6 of the CPSR
- The PC is forced to begin executing at the relevant vector address.

For example, if an interrupt for vector 0x18 occurs, the processor will start to execute code at address 0x18. The memory area that is used as start location for interrupts is called the interrupt vector table. The content of the interrupt vector is normally a branch instruction jumping to the interrupt routine.

Note: If the interrupt function enables interrupts, the special processor registers needed to return from the interrupt routine must be assumed to be destroyed. For this reason they must be stored by the interrupt routine to be restored before it returns. This is handled automatically if the `__nested` keyword is used.

INTERRUPTS FOR ARM CORTEX-M

ARM Cortex-M has a different interrupt mechanism than previous ARM architectures, which means the primitives provided by the compiler are also different.

On ARM Cortex-M, an interrupt service routine enters and returns in the same way as a normal function, which means no special keywords are required. Thus, the keywords `__irq`, `__fiq`, and `__nested` are not available when you compile for ARM Cortex-M.

These exception function names are defined in `cstartup_M.c` and `cstartup_M.s`. They are referred to by the library exception vector code:

```
NMI_Handler
HardFault_Handler
MemManage_Handler
BusFault_Handler
UsageFault_Handler
SVC_Handler
DebugMon_Handler
PendSV_Handler
SysTick_Handler
```

The vector table is implemented as an array. It should always have the name `__vector_table`, because `cmain` refers to that symbol and C-SPY looks for that symbol when determining where the vector table is located.

The predefined exception functions are defined as weak symbols. A weak symbol is only included by the linker as long as no duplicate symbol is found. If another symbol is defined with the same name, it will take precedence. Your application can therefore simply define its own exception function by just defining it using the correct name from the list above. If you need other interrupts or other exception handlers, you must make a copy of the `cstartup_M.c` or `cstartup_M.s` file and make the proper addition to the vector table.

The intrinsic functions `__get_CPSR` and `__set_CPSR` are not available when you compile for ARM Cortex-M. Instead, if you need to get or set values of these or other registers, you can use inline assembler. For more information, see *Passing values between C and assembler objects*, page 204.

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types, with the exception that interrupt member functions must be static. When a non-static member function is called, it must be applied to an object. When an interrupt occurs and the interrupt function is called, there is no object available to apply the member function to.

Special function types can be used for static member functions. For example, in the following example, the function `handler` is declared as an interrupt function:

```
class Device
{
    static __arm __irq void handler();
};
```


Linking using ILINK

This chapter describes the linking process using the IAR ILINK Linker and the related concepts—first with an overview and then in more detail.

Linking—an overview

The IAR ILINK Linker is a powerful, flexible software tool for use in the development of embedded applications. 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.

ILINK combines one or more relocatable object files—produced by the IAR Systems compiler or assembler—with selected parts of one or more object libraries to produce an executable image in the industry-standard format *Executable and Linking Format* (ELF).

ILINK will automatically load only those library modules—user libraries and Standard C or C++ library variants—that are actually needed by the application you are linking. Further, ILINK eliminates duplicate sections and sections that are not required.

ILINK can link both ARM and Thumb code, as well as a combination of them. By automatically inserting additional instructions (veneers), ILINK will assure that the destination will be reached for any calls and branches, and that the processor state is switched when required. For more details about how to generate veneers, see *Veneers*, page 90.

ILINK uses a *configuration file* where you can specify separate locations for code and data areas of your target system memory map. This file also supports automatic handling of the application's initialization phase, which means initializing global variable areas and code areas by copying initializers and possibly decompressing them as well.

The final output produced by ILINK is an absolute object file containing the executable image in the ELF (including DWARF for debug information) format. The file can be downloaded to C-SPY or any other compatible debugger that supports ELF/DWARF, or it can be stored in EPROM or flash.

To handle ELF files, various tools are included. For information about included utilities, see *Specific ELF tools*, page 32.

Modules and sections

Each relocatable object file contains one module, which consists of:

- Several sections of code or data
- Runtime attributes specifying various types of information, for example the version of the runtime environment
- Optionally, debug information in DWARF format
- A symbol table of all global symbols and all external symbols used.

A *section* is a logical entity containing a piece of data or code that should be placed at a physical location in memory. A section can consist of several *section fragments*, typically one for each variable or function (symbols). A section can be placed either in RAM or in ROM. In a normal embedded application, sections that are placed in RAM do not have any content, they only occupy space.

Each section has a name and a type attribute that determines the content. The type attribute is used (together with the name) for selecting sections for the ILINK configuration. The most commonly used attributes are:

code	Executable code
readonly	Constant variables
readwrite	Initialized variables
zeroinit	Zero-initialized variables

Note: In addition to these section types—sections that contain the code and data that are part of your application—a final object file will contain many other types of sections, for example sections that contain debugging information or other type of meta information.

A section is the smallest linkable unit; but if possible, ILINK can exclude smaller units—section fragments—from the final application. For more information, see *Keeping modules, page 85*, and *Keeping symbols and sections, page 85*.

At compile time, data and functions are placed in different sections. At link time, one of the most important functions of the linker is to assign addresses to the various sections used by the application.

The IAR build tools have many predefined section names. See the chapter *Section reference* for more information about each section.

The linking process

The relocatable modules in object files and libraries, produced by the IAR compiler and assembler, cannot be executed as is. To become an executable application, they must be *linked*.

Note: Modules produced by a toolset from another vendor can be included in the build as well, as long as the module is AEABI (ARM Embedded Application Binary Interface) compliant. Be aware that this also might require a compiler utility library from the same vendor.

The IAR ILINK Linker is used for the link process. It normally performs the following procedure (note that some of the steps can be turned off by command line options or by directives in the linker configuration file):

- Determine which modules to include in the application. Modules provided in object files are always included. A module in a library file is only included if it provides a definition for a global symbol that is referenced from an included module.
- Select which standard library files to use. The selection is based on attributes of the included modules. These libraries are then used for satisfying any still outstanding undefined symbols.
- Determine which sections/section fragments from the included modules to include in the application. Only those sections/section fragments that are actually needed by the application are included. There are several ways to determine of which sections/section fragments that are needed, for example, the `__root` object attribute, the `#pragma required` directive, and the `keep` linker directive. In case of duplicate sections, only one is included.
- Where appropriate, arrange for the initialization of initialized variables and code in RAM. The `initialize` directive causes the linker to create extra sections to enable copying from ROM to RAM. Each section that will be initialized by copying is divided into two sections, one for the ROM part and one for the RAM part. If manual initialization is not used, the linker also arranges for the startup code to perform the initialization.
- Determine where to place each section according to the section placement directives in the *linker configuration file*. Sections that are to be initialized by copying appear twice in the matching against placement directives, once for the ROM part and once for the RAM part, with different attributes. During the placement, the linker also adds any required veneers to make a code reference reach its destination or to switch CPU modes.
- Produce an absolute file that contains the executable image and any debug information provided. The contents of each needed section in the relocatable input files is calculated using the relocation information supplied in its file and the addresses determined when placing sections. This process can result in one or more

relocation failures if some of the requirements for a particular section are not met, for instance if placement resulted in the destination address for a PC-relative jump instruction being out of range for that instruction.

- Optionally, produce a map file that lists the result of the section placement, the address of each global symbol, and finally, a summary of memory usage for each module and library.

This illustration shows the linking process:

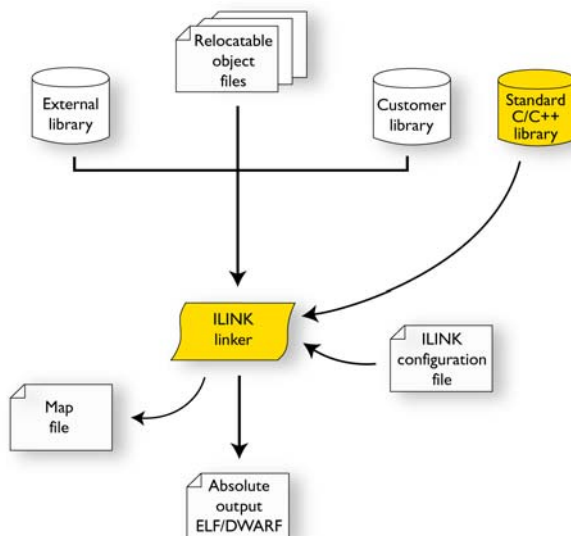


Figure 8: The linking process

During the linking, ILINK might produce error messages and logging messages on `stdout` and `stderr`. The log messages are useful for understanding why an application was linked as it was. For example, why a module or section (or section fragment) was included.

Note: To see the actual content of an ELF object file, use `ielddumparm`. See *The IAR ELF Dumper for ARM—ielddumparm*, page 422.

Placing code and data—the linker configuration file

The placement of sections in memory is performed by the IAR ILINK Linker. It uses the *linker configuration file* where you can define how ILINK should treat each section and how they should be placed into the available memories.

A typical linker configuration file contains definitions of:

- Available addressable memories
- Populated regions of those memories
- How to treat input sections
- Created sections
- How to place sections into the available regions.

The file consists of a sequence of declarative directives. This means that the linking process will be governed by all directives at the same time.

To use the same source code with different derivatives, just rebuild the code with the appropriate configuration file.

A SIMPLE EXAMPLE OF A CONFIGURATION FILE

A simple configuration file can look like this:

```
/* The memory space denoting the maximum possible amount
   of addressable memory */
define memory Mem with size = 4G;

/* Memory regions in an address space */
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* Create a stack */
define block STACK with size = 0x1000, alignment = 8 { };

/* Handle initialization */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* Initialize RW sections,
                                   exclude zero-initialized
                                   sections */

/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };

/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                             ROM: .rodata and .data_init */
```

```
place in RAM { readwrite, /* Place .data, .bss, and .noinit */
                  block STACK }; /* and STACK */
```

This configuration file defines one addressable memory `Mem` with the maximum of 4 Gbytes of memory. Further, it defines a ROM region and a RAM region in `Mem`, namely ROM and RAM. Each region has the size of 64 Kbytes.

The file then creates an empty block called `STACK` with a size of 4 Kbytes in which the application stack will reside. To create a *block* is the basic method which you can use to get detailed control of placement, size, etc. It can be used for grouping sections, but also as in this example, to specify the size and placement of an area of memory.

Next, the file defines how to handle the initialization of variables, read/write type (`readwrite`) sections. In this example, the initializers are placed in ROM and copied at startup of the application to the RAM area. By default, ILINK may compress the initializers if this appears to be advantageous.

The last part of the configuration file handles the actual placement of all the sections into the available regions. First, the startup code—defined to reside in the read-only (`readonly`) section `.cstartup`—is placed at the start of the ROM region, that is at address `0x10000`. Note that the part within `{ }` is referred to as *section selection* and it selects the sections for which the directive should be applied to. Then the rest of the read-only sections are placed in the ROM region. Note that the section selection `{ readonly section .cstartup }` takes precedence over the more generic section selection `{ readonly }`.

Finally, the read/write (`readwrite`) sections and the `STACK` block are placed in the RAM region.

This illustration gives a schematic overview of how the application is placed in memory:

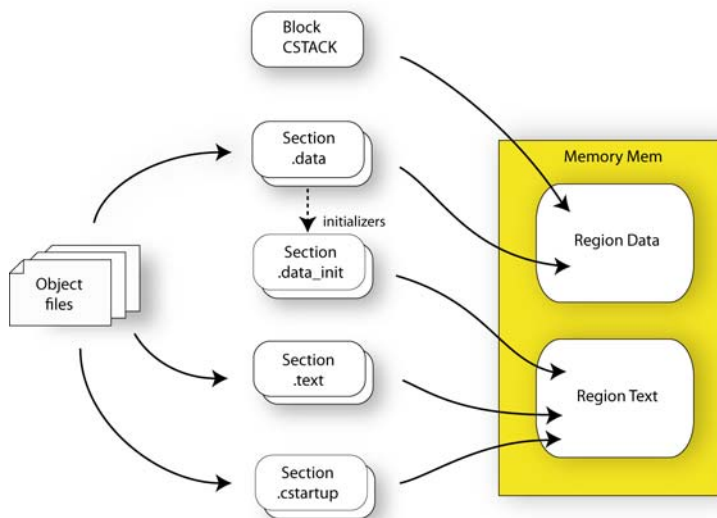


Figure 9: Application in memory

In addition to these standard directives, a configuration file can contain directives that define how to:

- Map a memory that can be addressed in multiple ways
- Handle conditional directives
- Create symbols with values that can be used in the application
- More in detail, select the sections a directive should be applied to
- More in detail, initialize code and data.

For more details and examples about customizing the linker configuration file, see the chapter *Linking your application*.

For more information about the linker configuration file, see the chapter *The linker configuration file*.

Initialization at system startup

In Standard C, all static variables—variables that are allocated at a fixed memory address—must be initialized by the runtime system to a known value at application startup. This value is either an explicit value assigned to the variable, or if no value is given, it is cleared to zero. In the compiler, there is one exception to this rule and that is variables declared `__no_init` which are not initialized at all.

The compiler generates a specific type of section for each type of variable initialization:

Categories of declared data	Source	Section type	Section name	Section content
Zero-initialized data	<code>int i;</code>	Read/write data, zero-init	<code>.bss</code>	None
Zero-initialized data	<code>int i = 0;</code>	Read/write data, zero-init	<code>.bss</code>	None
Initialized data (non-zero)	<code>int i = 6;</code>	Read/write data	<code>.data</code>	The initializer
Non-initialized data	<code>__no_init int i;</code>	Read/write data, zero-init	<code>.noinit</code>	None
Constants	<code>const int i = 6;</code>	Read-only data	<code>.rodata</code>	The constant
Code	<code>__ramfunc void myfunc() {}</code>	Read/write code	<code>.textrw</code>	The code

Table 3: Sections holding initialized data

Note: Clustering of static variables might group zero-initialized variables together with initialized data in `.data`. The compiler can decide to place constants in the `.text` section to avoid loading the address of a constant from a constant table.

For information about all supported sections, see the chapter *Section reference*.

THE INITIALIZATION PROCESS

Initialization of data is handled by ILINK and the system startup code in conjunction.

To configure the initialization of variables, you must consider these issues:

- Sections that should be zero-initialized are handled automatically by ILINK; they should only be placed in RAM

- Sections that should be initialized, except for zero-initialized sections, should be listed in an `initialize` directive

Normally during linking, a section that should be initialized is split into two sections, where the original initialized section will keep the name. The contents are placed in the new initializer section, which will get the original name suffixed with `_init`. The initializers should be placed in ROM and the initialized sections in RAM, by means of placement directives. The most common example is the `.data` section which the linker splits into `.data` and `.data_init`.

- Sections that contains constants should not be initialized; they should only be placed in flash/ROM
- Sections holding `__no_init` declared variables should not be initialized and thus should be listed in a `do not initialize` directive. They should also be placed in RAM.

In the linker configuration file, it can look like this:

```
/* Handle initialization */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* Initialize RW sections,
                                   exclude zero-initialized
                                   sections */

/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };

/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                             ROM: .rodata and .data_init */
place in RAM { readwrite, /* Place .data, .bss, and .noinit */
               block STACK }; /* and STACK */
```

Note: When compressed initializers are used (see *Initialize directive*, page 392), the contents sections (that is, sections with the `_init` suffix) are not listed as separate sections in the map file. Instead, they are combined into aggregates of “initializer bytes”. You can place the contents sections the usual way in the linker configuration file; however, this affects the placement (and possibly the number) of the “initializer bytes” aggregates.

For more information about and examples of how to configure the initialization, see *Linking considerations*, page 81.

C++ DYNAMIC INITIALIZATION

The compiler places subroutine pointers for performing C++ dynamic initialization into sections of the ELF section types `SHT_PREINIT_ARRAY` and `SHT_INIT_ARRAY`. By default, the linker will place these into a linker-created block, ensuring that all sections of the section type `SHT_PREINIT_ARRAY` are placed before those of the type `SHT_INIT_ARRAY`. If any such sections were included, code to call the routines will also be included.

The linker-created blocks are only generated if the linker configuration does not contain section selector patterns for the `preinit_array` and `init_array` section types. The effect of the linker-created blocks will be very similar to what happens if the linker configuration file contains this:

```
define block SHT$$PREINIT_ARRAY { preinit_array };
define block SHT$$INIT_ARRAY { init_array };
define block CPP_INIT with fixed order { block
                                     SHT$$PREINIT_ARRAY,
                                     block SHT$$INIT_ARRAY };
```

If you put this into your linker configuration file, you must also mention the `CPP_INIT` block in one of the section placement directives. If you wish to select where the linker-created block is placed, you can use a section selector with the name `".init_array"`.

See also *Section-selectors*, page 397.

Stack usage analysis

Under the right circumstances, the linker can accurately calculate the maximum stack usage for each call graph root (each function that is not called from another function).

If you enable stack usage analysis, a stack usage chapter will be added to the linker map file, listing for each call graph root the particular call chain which results in the maximum stack depth.

This is only accurate if there is accurate stack usage information for each function in the application.

In general, the compiler will generate this information for each C function, but if there are indirect calls (calls using function pointers) in your application, you must supply a list of possible functions that can be called from each calling function. You can do this by using pragma directives in the source file, or by using a separate stack usage control file when linking.

If you use a stack usage control file, you can also supply stack usage information for functions in modules that do not have stack usage information.

SITUATIONS WHERE WARNINGS ARE ISSUED

When stack usage analysis is enabled in the linker, warnings will be generated in the following circumstances:

- There is at least one function without stack usage information.
- There is at least one indirect call site in the application for which a list of possible called functions has not been supplied.
- There are no known indirect calls, but there is at least one uncalled function that is not known to be a call graph root.
- The application contains recursion (a cycle in the call graph).
- There are calls to a function declared as a call graph root.

LIMITATIONS

Apart from missing or incorrect stack usage information, there are also other sources of inaccuracy in the analysis:

- The linker might not always be able to identify all functions in object modules that lack stack usage information.
- If you use inline assembler to change the frame size or to perform function calls, this will not be reflected in the analysis.
- Extra space consumed by other sources (the processor, an operating system, etc) is not accounted for.
- C++ source code is, in general, not supported. In particular, virtual function calls and exceptions are not handled.
- If you use other forms of function calls, like software interrupts, they will not be reflected in the call graph.

Note that stack usage analysis produces a worst case result. The application might not actually ever end up in the maximum call chain, by design or by coincidence.

The linker does not check that you have allocated enough stack, nor is there a way to incorporate the results of stack usage analysis in the link configuration.



Stack usage analysis is only a complement to actual measurement. If the result is important, you need to perform independent validation of the figures arrived at through analysis.

STACK USAGE CONTROL FILES

A stack usage control file contains stack usage control directives.

Using stack usage control files, you can:

- Specify complete stack usage information (call graph root category, stack usage, and possible calls) for a function, by using the stack usage control directive `function`.
- Exclude certain functions from stack usage analysis, by using the stack usage control directive `exclude`.
- Specify the possible destinations for indirect calls in a function, by using the stack usage control directive `possible calls`.
- Specify that functions are call graph roots, including an optional call graph root category, by using the stack usage control directive `call graph root`.

If your interrupt functions have not already been designated as call graph roots by the compiler, you must do so manually. You can do this either by using the `#pragma call_graph_root` directive in your source code or by using a simple stack usage control file, which might look something like this:

```
call graph root [interrupt]: Irq1Handler, Irq2Handler;
```

For more information, see *call_graph_root*, page 320 and the chapter *Stack usage control files*, page 411.

SOURCE ANNOTATION

In C files, at the point of an indirect call, you can use the `#pragma calls` directive to list the possible destinations for that call.

You can also, at the definition of a function, specify that it is a call graph root by using the `#pragma call_graph_root` directive.

MAP FILE CONTENTS

When stack usage analysis is enabled, the linker map file contains a stack usage chapter that lists the call chain that results in the maximum stack depth for each call graph root. This is an example of what the stack usage chapter in the map file might look like:

```
*****
*****
***  STACK USAGE
***

Entry
__iar_program_start: 0x0000c909
```

Maximum call chain	576 bytes
__iar_program_start	0
__cmain	0
main	32
aequals	24
errmsg [util.o(libutil.a)]	16
pr_l	80
pr_ok	16
printf	24
_PrintfFullNoMb	160
_PutcharsFullNoMb [xprintffull_nomb.o(dl6M_tlf.a)]	24
_Prout	16
putchar	8
fputc	16
_Fwprep	16
fseek	16
_Fspos	24
fflush	24
fflushOne [fflush.o(dl6M_tlf.a)]	16
__write	8
__dwrite	8
__iar_sh_stdout	24
__iar_get_ttio	24
__iar_lookup_ttioh	0

In this case, the maximum stack depth for the program entry (`__iar_program_entry`) is 576 bytes, and occurs inside the system library `printf` function. Public functions are listed by name, while module-local functions also include the name of the module (like `errmsg` above).

When you have several call graph roots, for example if there are a number of interrupt functions, you must combine the stack depths from the different roots appropriately, considering the fact that they might need space on different stacks.

CALL GRAPH OUTPUT

The linker can also produce a call graph file in XML format. This file contains one node for each function in your application, with the stack usage and call information relevant to that function. It is intended to be input for post-processing tools and is not particularly human-readable.

For more information about the XML format used, see the `callGraph.txt` file in your product installation.

Linking your application

This chapter lists aspects that you must consider when linking your application. This includes using ILINK options and tailoring the linker configuration file.

Finally, this chapter provides some hints for troubleshooting.

Linking considerations

Before you can link your application, you must set up the configuration required by ILINK. Typically, you must consider:

- Defining your own memory areas
- Placing sections
- Keeping modules in the application
- Keeping symbols and sections in the application
- Application startup
- Setting up the stack and heap
- Setting up the `atexit` limit
- Changing the default initialization
- Symbols for controlling the application
- Standard library handling
- Other output formats than ELF/DWARF
- Veneers.

CHOOSING A LINKER CONFIGURATION FILE

The `config` directory contains two ready-made templates for the linker configuration file:

- `generic.icf`, designed for all cores except for Cortex-M cores
- `generic_cortex.icf`, designed for all Cortex-M cores.

These files contain the information required by ILINK. The only change, if any, you will normally have to make to the supplied configuration file is to customize the start and end addresses of each region so they fit the target system memory map. If, for example, your application uses additional external RAM, you must also add details about the external RAM memory area.

To edit a linker configuration file, use the editor in the IDE, or any other suitable editor. Alternatively, choose **Project>Options>Linker** and click the **Edit** button on the **Config** page to open the dedicated linker configuration file editor.

Remember not to change the original template file. We recommend that you make a copy in the working directory, and modify the copy instead. If you are using the linker configuration file editor in the IDE, the IDE will make a copy for you.

Each project in the IDE should have a reference to one, and only one, linker configuration file. This file can be edited, but for the majority of all projects it is sufficient to configure the vital parameters in **Project>Options>Linker>Config**.

DEFINING YOUR OWN MEMORY AREAS

The default configuration file that you selected has predefined ROM and RAM regions. This example will be used as a starting-point for all further examples in this chapter:

```
/* Define the addressable memory */
define memory Mem with size = 4G;

/* Define a region named ROM with start address 0 and to be 64
Kbytes large */
define region ROM = Mem:[from 0 size 0x10000];

/* Define a region named RAM with start address 0x20000 and to be
64 Kbytes large */
define region RAM = Mem:[from 0x20000 size 0x10000];
```

Each region definition must be tailored for the actual hardware.

To find out how much of each memory that was filled with code and data after linking, inspect the memory summary in the map file (command line option `--map`).

Adding an additional region

To add an additional region, use the `define region` directive, for example:

```
/* Define a 2nd ROM region to start at address 0x80000 and to be
128 Kbytes large */
define region ROM2 = Mem:[from 0x80000 size 0x20000];
```

Merging different areas into one region

If the region is comprised of several areas, use a region expression to merge the different areas into one region, for example:

```
/* Define the 2nd ROM region to have two areas. The first with
the start address 0x80000 and 128 Kbytes large, and the 2nd with
the start address 0xC0000 and 32 Kbytes large */
```

```
define region ROM2 = Mem:[from 0x80000 size 0x20000]
                    | Mem:[from 0xC0000 size 0x08000];
```

or equivalently

```
define region ROM2 = Mem:[from 0x80000 to 0xC7FFF]
                    -Mem:[from 0xA0000 to 0xBFFFF];
```

PLACING SECTIONS

The default configuration file that you selected places all predefined sections in memory, but there are situations when you might want to modify this. For example, if you want to place the section that holds constant symbols in the `CONSTANT` region instead of in the default place. In this case, use the `place in` directive, for example:

```
/* Place sections with readonly content in the ROM region */
place in ROM {readonly};
/* Place the constant symbols in the CONSTANT region */
place in CONSTANT {readonly section .rodata};
```

Note: Placing a section—used by the IAR build tools—in a different memory which use a different way of referring to its content, will fail.

For the result of each placement directive after linking, inspect the placement summary in the map file (the command line option `--map`).

Placing a section at a specific address in memory

To place a section at a specific address in memory, use the `place at` directive, for example:

```
/* Place section .vectors at address 0 */
place at address Mem:[0] {readonly section .vectors};
```

Placing a section first or last in a region

To place a section first or last in a region is similar, for example:

```
/* Place section .vectors at start of ROM */
place at start of ROM {readonly section .vectors};
```

Declare and place your own sections

To declare new sections—in addition to the ones used by the IAR build tools—to hold specific parts of your code or data, use mechanisms in the compiler and assembler. For example:

```
/* Create a section for variables. */
#pragma section = "MYOWNSECTION"

/* Place a variable in that section. */
const short MyVariable @ "MYOWNSECTION" = 0xF0F0;
```

This is the corresponding example in assembler language:

```
name      createSection
section MYOWNSECTION:CONST ; Create a section,
                                ; and fill it with
dc16      0xF0F0            ; constant bytes.
end
```

To place your new section, the original place in ROM {readonly}; directive is sufficient.

However, to place the section `MyOwnSection` explicitly, update the linker configuration file with a place in directive, for example:

```
/* Place MyOwnSection in the ROM region */
place in ROM {readonly section MyOwnSection};
```

RESERVING SPACE IN RAM

Often, an application must have an empty uninitialized memory area to be used for temporary storage, for example a heap or a stack. It is easiest to achieve this at link time. You must create a block with a specified size and then place it in a memory.

In the linker configuration file, it can look like this:

```
define block TempStorage with size = 0x1000, alignment = 4 { };
place in RAM { block TempStorage };
```

To retrieve the start of the allocated memory from the application, the source code could look like this:

```
/* Declare a section for temporary storage. */
#pragma section = "TEMPSTORAGE"

char *GetTempStorageStartAddress()
{
    /* Return start address of section TEMPSTORAGE. */
    return __section_begin("TEMPSTORAGE");
}
```

KEEPING MODULES

If a module is linked as an object file, it is always kept. That is, it will contribute to the linked application. However, if a module is part of a library, it is included only if it is symbolically referred to from other parts of the application. This is true, even if the library module contains a root symbol. To assure that such a library module is always included, use `iarchive` to extract the module from the library, see *The IAR Archive Tool—iarchive*, page 417.

For information about included and excluded modules, inspect the log file (the command line option `--log modules`).

For more information about modules, see *Modules and sections*, page 68.

KEEPING SYMBOLS AND SECTIONS

By default, ILINK removes any sections, section fragments, and global symbols that are not needed by the application. To retain a symbol that does not appear to be needed—or actually, the section fragment it is defined in—you can either use the root attribute on the symbol in your C/C++ or assembler source code, or use the ILINK option `--keep`. To retain sections based on attribute names or object names, use the directive `keep` in the linker configuration file.

To prevent ILINK from excluding sections and section fragments, use the command line options `--no_remove` or `--no_fragments`, respectively.

For information about included and excluded symbols and sections, inspect the log file (the command line option `--log sections`).

For more information about the linking procedure for keeping symbols and sections, see *The linking process*, page 69.

APPLICATION STARTUP

By default, the point where the application starts execution is defined by the `__iar_program_start` label, which is defined to point at the start of the `cstartup.s` file. The label is also communicated via ELF to any debugger that is used.

To change the start point of the application to another label, use the ILINK option `--entry`; see *--entry*, page 270.

SETTING UP THE STACK

The size of the `CSTACK` block is defined in the linker configuration file. To change the allocated amount of memory, change the block definition for `CSTACK`:

```
define block CSTACK with size = 0x2000, alignment = 8{ };
```

Specify an appropriate size for your application.

For more information about the stack, see *Stack considerations*, page 173.

SETTING UP THE HEAP

The size of the heap is defined in the linker configuration file as a block:

```
define block HEAP with size = 0x1000, alignment = 8{ };
place in RAM {block HEAP};
```

Specify the appropriate size for your application.

SETTING UP THE ATEXIT LIMIT

By default, the `atexit` function can be called a maximum of 32 times from your application. To either increase or decrease this number, add a line to your configuration file. For example, to reserve room for 10 calls instead, write:

```
define symbol __iar_maximum_atexit_calls = 10;
```

CHANGING THE DEFAULT INITIALIZATION

By default, memory initialization is performed during application startup. ILINK sets up the initialization process and chooses a suitable packing method. If the default initialization process does not suit your application and you want more precise control over the initialization process, these alternatives are available:

- Choosing the packing algorithm
- Manual initialization
- Initializing code—copying ROM to RAM.

For information about the performed initializations, inspect the log file (the command line option `--log initialization`).

Choosing packing algorithm

To override the default packing algorithm, write for example:

```
initialize by copy with packing = lzw { readwrite };
```

For more information about the available packing algorithms, see *Initialize directive*, page 392.

Manual initialization

The `initialize manually` directive lets you take complete control over initialization. For each involved section, ILINK creates an extra section that contains the initialization data, but makes no arrangements for the actual copying. This directive is, for example, useful for overlays:

```
/* Sections MYOVERLAY1 and MYOVERLAY2 will be overlaid in
MyOverlay */
define overlay MyOverlay { section MYOVERLAY1 };
define overlay MyOverlay { section MYOVERLAY2 };

/* Split the overlay sections but without initialization during
system startup */
initialize manually { section MYOVERLAY* };

/* Place the initializer sections in a block each */
define block MyOverlay1InRom { section MYOVERLAY1_init };
define block MyOverlay2InRom { section MYOVERLAY2_init };

/* Place the overlay and the initializers for it */
place in RAM { overlay MyOverlay };
place in ROM { block MyOverlay1InRom, block MyOverlay2InRom };
```

The application can then start a specific overlay by copying, as in this case, ROM to RAM:

```
#include <string.h>

/* Declare the overlay sections. */

#pragma section = "MYOVERLAY"
#pragma section = "MYOVERLAY1INROM"

/* Function that switches in image 1 into the overlay. */

void SwitchToOverlay1()
{
    char *targetAddr      = __section_begin("MYOVERLAY");
    char *sourceAddr      = __section_begin("MYOVERLAY1INROM");
    char *sourceAddrEnd   = __section_end("MYOVERLAY1INROM");
    int size = sourceAddrEnd - sourceAddr;

    memcpy(targetAddr, sourceAddr, size);
}
```

Initializing code—copying ROM to RAM

Sometimes, an application copies pieces of code from flash/ROM to RAM. This can be easily achieved by ILINK for whole code regions. However, for individual functions, the `__ramfunc` keyword can be used, see *Execution in RAM*, page 58.

List the code sections that should be initialized in an `initialize` directive and then place the initializer and initialized sections in ROM and RAM, respectively.

In the linker configuration file, it can look like this:

```
/* Split the .textrw section into a readonly and a readwrite
section */
initialize by copy { section .textrw };

/* Place both in a block */
define block RamCode { section .textrw };
define block RamCodeInit { section .textrw_init };

/* Place them in ROM and RAM */
place in ROM { block RamCodeInit };
place in RAM { block RamCode };
```

The block definitions makes it possible to refer to the start and end of the blocks from the application.

For more examples, see *Interaction between the tools and your application*, page 176.

Running all code from RAM

If you want to copy the entire application from ROM to RAM at program startup, use the `initilize by copy` directive, for example:

```
initialize by copy { readonly, readwrite };
```

The `readwrite` pattern will match all statically initialized variables and arrange for them to be initialized at startup. The `readonly` pattern will do the same for all read-only code and data, except for code and data needed for the initialization.

To reduce the ROM space that is needed, it might be useful to compress the data with one of the available packing algorithms. For example,

```
initialize by copy with packing = lzw { readonly, readwrite };
```

For more information about the available compression algorithms, see *Initialize directive*, page 392.

Because the function `__low_level_init`, if present, is called before initialization, it, and anything it needs, will not be copied from ROM to RAM either. In some circumstances—for example, if the ROM contents are no longer available to the program after startup—you might need to avoid using the same functions during startup and in the rest of the code.

If anything else should not be copied, include it in an `except` clause. This can apply to, for example, the interrupt vector table.

It is also recommended to exclude the C++ dynamic initialization table from being copied to RAM, as it is typically only read once and then never referenced again. For example, like this:

```
initialize by copy { readonly, readwrite }
    except { section .intvec,          /* Don't copy
                                         interrupt table */
            section .init_array }; /* Don't copy
                                         C++ init table */
```

INTERACTION BETWEEN ILINK AND THE APPLICATION

ILINK provides the command line options `--config_def` and `--define_symbol` to define symbols which can be used for controlling the application. You can also use symbols to represent the start and end of a continuous memory area that is defined in the linker configuration file. For more information, see *Interaction between the tools and your application*, page 176.

To change a reference to one symbol to another symbol, use the ILINK command line option `--redirect`. This is useful, for example, to redirect a reference from a non-implemented function to a stub function, or to choose one of several different implementations of a certain function, for example, how to choose the DLIB formatter for the standard library functions `printf` and `scanf`.

The compiler generates mangled names to represent complex C/C++ symbols. If you want to refer to these symbols from assembler source code, you must use the mangled names.

For information about the addresses and sizes of all global (statically linked) symbols, inspect the entry list in the map file (the command line option `--map`).

For more information, see *Interaction between the tools and your application*, page 176.

STANDARD LIBRARY HANDLING

By default, ILINK determines automatically which variant of the standard library to include during linking. The decision is based on the sum of the runtime attributes available in each object file and the library options passed to ILINK.

To disable the automatic inclusion of the library, use the option `--no_library_search`. In this case, you must explicitly specify every library file to be included. For information about available library files, see *Using prebuilt libraries*, page 95.

PRODUCING OTHER OUTPUT FORMATS THAN ELF/DWARF

ILINK can only produce an output file in the ELF/DWARF format. To convert that format into a format suitable for programming PROM/flash, see *The IAR ELF Tool—ielftool*, page 420.

VENEERS

The ARM cores need to use veneers on two occasions:

- When calling an ARM function from Thumb mode or vice versa; the veneer then changes the state of the microprocessor. If the core supports the `BLX` instruction, a veneer is not needed for changing modes.
- When calling a function that it cannot normally reach; the veneer introduces code which makes the call successfully reach the destination.

Code for veneers can be inserted between any caller and called function. As a result, the `R12` register must be treated as a scratch register at function calls, including functions written in assembler. This also applies to jumps.

For more information, see `--no_veneers`, page 280.

Hints for troubleshooting

ILINK has several features that can help you manage code and data placement correctly, for example:

- Messages at link time, for examples when a relocation error occurs
- The `--log` option that makes ILINK log information to `stdout`, which can be useful to understand why an executable image became the way it is, see `--log`, page 275
- The `--map` option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see `--map`, page 276.

RELOCATION ERRORS

For each instruction that cannot be relocated correctly, ILINK will generate a *relocation error*. This can occur for instructions where the target is out of reach or is of an incompatible type, or for many other reasons.

A relocation error produced by ILINK can look like this:

```
Error[Lp002]: relocation failed: out of range or illegal value
  Kind      :  R_XXX_YYY[0x1]
  Location  :  0x40000448
               "myfunc" + 0x2c
               Module:  somecode.o
               Section: 7 (.text)
               Offset:  0x2c
  Destination: 0x9000000c
               "read"
               Module:  read.o(iolib.a)
               Section: 6 (.text)
               Offset:  0x0
```

The message entries are described in this table:

Message entry	Description
Kind	The relocation directive that failed. The directive depends on the instruction used.
Location	<p>The location where the problem occurred, described with the following details:</p> <ul style="list-style-type: none"> • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, 0x40000448 and "myfunc" + 0x2c. • The module, and the file. In this example, the module <code>somecode.o</code>. • The section number and section name. In this example, section number 7 with the name <code>.text</code>. • The offset, specified in number of bytes, in the section. In this example, 0x2c.

Table 4: Description of a relocation error

Message entry	Description
Destination	<p>The target of the instruction, described with the following details:</p> <ul style="list-style-type: none">• The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, 0x9000000c and "read" (thus, no offset).• The module, and when applicable the library. In this example, the module read.o and the library iolib.a.• The section number and section name. In this example, section number 6 with the name .text.• The offset, specified in number of bytes, in the section. In this example, 0x0.

Table 4: Description of a relocation error (Continued)

Possible solutions

In this case, the distance from the instruction in myfunc to __read is too long for the branch instruction.

Possible solutions include ensuring that the two .text sections are allocated closer to each other 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 section placement.

The DLIB runtime environment

This chapter describes the runtime environment in which an application executes. In particular, the chapter covers the DLIB runtime library and how you can optimize it for your application.

Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code.

RUNTIME ENVIRONMENT FUNCTIONALITY

The *runtime environment* supports Standard C and C++, including the standard template library. The runtime environment consists of the *runtime library*, which contains the functions defined by the C and the C++ standards, and include files that define the library interface (the system header files).

The runtime library is delivered both as prebuilt libraries and (depending on your product package) as source files, and you can find them in the product subdirectories `arm\lib` and `arm\src\lib`, respectively.

The runtime environment also consists of a part with specific support for the target system, which includes:

- Support for hardware features:
 - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
 - Peripheral unit registers and interrupt definitions in include files
 - The Vector Floating Point (VFP) coprocessor.
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- A floating-point environment (*fenv*) that contains floating-point arithmetics support, see *fenv.h*, page 379.
- Special compiler support, for instance functions for switch handling or integer arithmetics.

For information about AEABI compliance, see *AEABI compliance*, page 182.

For more information about the library, see the chapter *Library functions*.

SETTING UP THE RUNTIME ENVIRONMENT

The IAR DLIB runtime environment can be used as is together with the debugger. However, to run the application on hardware, you must adapt the runtime environment. Also, to configure the most code-efficient runtime environment, you must determine your application and hardware requirements. The more functionality you need, the larger your code will become.

This is an overview of the steps involved in configuring the most efficient runtime environment for your target hardware:

- Choose which runtime library object file to use
It is not necessary to specify a library file explicitly, as ILINK automatically uses the correct library file. See *Using prebuilt libraries*, page 95.
- Choose which predefined runtime library configuration to use—Normal or Full
You can configure the level of support for certain library functionality, for example, locale, file descriptors, and multibyte characters. If you do not specify anything, a default library configuration file that matches the library object file is automatically used. To specify a library configuration explicitly, use the `--dlib_config` compiler option. See *Library configurations*, page 112.
- Optimize the size of the runtime library
You can specify the formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 99. You can also specify the size and placement of the stacks and the heap, see *Setting up the stack*, page 85, and *Setting up the heap*, page 86, respectively.
- Include debug support for runtime and I/O debugging
The library offers support for mechanisms like redirecting standard input and output to the C-SPY Terminal I/O window and accessing files on the host computer, see *Application debug support*, page 101.
- Adapt the library for target hardware
The library uses a set of low-level functions for handling accesses to your target system. To make these accesses work, you must implement your own version of these functions. For example, to make `printf` write to an LCD display on your board, you must implement a target-adapted version of the low-level function `__write`, so that it can write characters to the display. To customize such functions, you need a good understanding of the library low-level interface, see *Adapting the library for target hardware*, page 104.

- Override library modules

If you have customized the library functionality, you need to make sure your versions of the library modules are used instead of the default modules. This can be done without rebuilding the entire library, see *Overriding library modules*, page 105.

- Customize system initialization

It is likely that you need to customize the source code for system initialization, for example, your application might need to initialize memory-mapped special function registers, or omit the default initialization of data sections. You do this by customizing the routine `__low_level_init`, which is executed before the data sections are initialized. See *System startup and termination*, page 107 and *Customizing system initialization*, page 111.

- Configure your own library configuration files

In addition to the prebuilt library configurations, you can make your own library configuration, but that requires that you *rebuild* the library. This gives you full control of the runtime environment. See *Building and using a customized library*, page 105.

- Manage a multithreaded environment

In a multithreaded environment, you must adapt the runtime library to treat all library objects according to whether they are global or local to a thread. See *Managing a multithreaded environment*, page 122.

- Check module consistency

You can use runtime model attributes to ensure that modules are built using compatible settings, see *Checking module consistency*, page 127.

Using prebuilt libraries

The prebuilt runtime libraries are configured for different combinations of these features:

- Architecture
- CPU mode
- Byte order
- Library configuration—Normal or Full
- Floating-point implementation.

The linker will automatically include the correct library object file and library configuration file. To explicitly specify a library configuration, use the `--dlib_config` option. For more information, see *Runtime environment*, page 50.

LIBRARY FILENAME SYNTAX

The names of the libraries are constructed by these constituents:

<code>{architecture}</code>	is the name of the architecture. It can be one of <code>4t</code> , <code>5E</code> , <code>6M</code> , or <code>7M</code> for the ARM architectures <code>v4T</code> , <code>v5TE</code> , <code>v6M</code> , or <code>v7M</code> , respectively. Libraries built for the <code>v5TE</code> architecture are also used for the <code>v6</code> architecture and later (except for <code>v6M</code> and <code>v7M</code>).
<code>{cpu_mode}</code>	is one of <code>t</code> or <code>a</code> , for Thumb and ARM, respectively.
<code>{byte_order}</code>	is one of <code>l</code> or <code>b</code> , for little-endian and big-endian, respectively.
<code>{fp_implementation}</code>	<p><code>_</code> when the library is compiled without VFP support, that is, using a software implementation floating-point functions.</p> <p><code>v</code> when the library is compiled with VFP support for architectures VFPv2 or later. Libraries compiled with VFP support have two entries for each function with floating-point signature. One entry is compliant with the VFP variant of AAPCS, the other is compliant with the AAPCS base standard. The linker will use the VFP variant entry for modules compiled with the VFP calling convention, and the base standard entry for other modules.</p>
<code>{language}</code>	is <code>c</code> when the library is compiled for Standard C++ support, and <code>e</code> when compiled for Embedded C++ support.
<code>{lib_config}</code>	is one of <code>n</code> or <code>f</code> for normal and full, respectively.
<code>{debug_interface}</code>	is one of <code>s</code> , <code>b</code> or <code>i</code> , for the SWI/SVC mechanism, the BKPT mechanism, and the IAR-specific breakpoint mechanism, respectively. For more information, see <i>--semihosting</i> , page 283.
<code>{rwpi}</code>	is <code>s</code> when the library contains read/write position-independent code, see <i>--rwpi</i> , page 255.

Note: There are two library configuration files: `DLib_Config_Normal.h` and `DLib_Config_Full.h`.

You can find the library object files and the library configuration files in the subdirectory `arm\lib`.

GROUPS OF LIBRARY FILES

The libraries are delivered in groups of library functions.

Library files for C library functions

These are the functions defined by Standard C, for example functions like `printf` and `scanf`. Note that this library does not include math functions.

The names of the library files are constructed in the following way:

```
dl<architecture>_<cpu_mode><byte_order><lib_config><rwpi>.a
```

which more specifically means

```
dl<4t|5E|6M|7M>_<a|t><l|b><n|f><s>.a
```

Library files for C++ and Embedded C++ library functions

These are the functions defined by C++, compiled with support for either Standard C++ or Embedded C++.

The names of the library files are constructed in the following way:

```
dlpp<architecture>_<cpu_mode><byte_order><fp_implementation>  
<lib_config><language>.a
```

which more specifically means

```
dlpp<4t|5E|6M|7M>_<a|t><l|b><_|v><n|f><c|e>.a
```

Library files for math functions

These are the functions for floating-point arithmetic and functions with a floating-point type in its signature as defined by Standard C, for example functions like `sqrt`.

The names of the library files are constructed in the following way:

```
m<architecture>_<cpu_mode><byte_order><fp_implementation>.a
```

which more specifically means

```
m<4t|5E|6M|7M>_<a|t><l|b><|v>.a
```

Library files for runtime support functions

These are functions for system startup, initialization, non floating-point AEABI support routines, and some of the functions part of Standard C and C++.

The names of the library files are constructed in the following way:

```
rt<architecture>_<cpu_mode><byte_order>.a
```

which more specifically means

```
rt<4t|5E|6M|7M>_<a|t><l|b>.a
```

Library files for debug support functions

These are functions for debug support for the semihosting interface. The names of the library files are constructed in the following way:

```
sh<debug_interface>_<byte_order>.a
```

which more specifically means

```
sh<s|b|i>_<l|b>.a
```

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the compiler can be used as is. However, you can customize parts of a library without rebuilding it.

These items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	<i>Choosing formatters for printf and scanf, page 99</i>
Startup and termination code	<i>System startup and termination, page 107</i>
Low-level input and output	<i>Standard streams for input and output, page 113</i>
File input and output	<i>File input and output, page 117</i>
Low-level environment functions	<i>Environment interaction, page 120</i>
Low-level signal functions	<i>Signal and raise, page 121</i>
Low-level time functions	<i>Time, page 121</i>
Size of heaps, stacks, and sections	<i>Stack considerations, page 173</i>
	<i>Heap considerations, page 175</i>
	<i>Placing code and data—the linker configuration file, page 71</i>

Table 5: Customizable items

For information about how to override library modules, see *Overriding library modules, page 105*.

Choosing formatters for printf and scanf

The linker automatically chooses an appropriate formatter for `printf`- and `scanf`-related function based on information from the compiler. If that information is missing or insufficient, for example if `printf` is used through a function pointer, if the object file is old, etc, then the automatic choice is the Full formatter. In this case you might want to choose a formatter manually.

To override the default formatter for all the `printf`- and `scanf`-related functions, except for `wprintf` and `wscanf` variants, you simply set the appropriate library options. This section describes the different options available.

Note: If you rebuild the library, you can optimize these functions even further, see *Configuration symbols for printf and scanf*, page 115.

CHOOSING A PRINTF FORMATTER

The `printf` function uses a formatter called `_Printf`. The full version is quite large, and provides facilities not required in many embedded applications. To reduce the memory consumption, three smaller, alternative versions are also provided in the Standard C/EC++ library.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	Tiny	Small/ SmallNoMb	Large/ LargeNoMb	Full/ FullNoMb
Basic specifiers <code>c</code> , <code>d</code> , <code>i</code> , <code>o</code> , <code>p</code> , <code>s</code> , <code>u</code> , <code>X</code> , <code>x</code> , and <code>%</code>	Yes	Yes	Yes	Yes
Multibyte support	No	Yes/No	Yes/No	Yes/No
Floating-point specifiers <code>a</code> , and <code>A</code>	No	No	No	Yes
Floating-point specifiers <code>e</code> , <code>E</code> , <code>f</code> , <code>F</code> , <code>g</code> , and <code>G</code>	No	No	Yes	Yes
Conversion specifier <code>n</code>	No	No	Yes	Yes
Format flag <code>+</code> , <code>-</code> , <code>#</code> , <code>0</code> , and space	No	Yes	Yes	Yes
Length modifiers <code>h</code> , <code>l</code> , <code>L</code> , <code>s</code> , <code>t</code> , and <code>Z</code>	No	Yes	Yes	Yes
Field width and precision, including <code>*</code>	No	Yes	Yes	Yes
<code>long long</code> support	No	No	Yes	Yes

Table 6: Formatters for printf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 115.



Specifying the print formatter in the IDE

To explicitly specify a formatter, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



Specifying the printf formatter from the command line

To explicitly specify a formatter, use one of these ILINK command line options:

```
--redirect _Printf=_PrintfFull
--redirect _Printf=_PrintfFullNoMb
--redirect _Printf=_PrintfLarge
--redirect _Printf=_PrintfLargeNoMb
--redirect _Printf=_PrintfSmall
--redirect _Printf=_PrintfSmallNoMb
--redirect _Printf=_PrintfTiny
--redirect _Printf=_PrintfTinyNoMb
```

CHOOSING A SCANF FORMATTER

In a similar way to the `printf` function, `scanf` uses a common formatter, called `_Scanf`. The full version is quite large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided in the Standard C/C++ library.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	Small/ SmallNoMb	Large/ LargeNoMb	Full/ FullNoMb
Basic specifiers <code>c</code> , <code>d</code> , <code>i</code> , <code>o</code> , <code>p</code> , <code>s</code> , <code>u</code> , <code>X</code> , <code>x</code> , and <code>%</code>	Yes	Yes	Yes
Multibyte support	Yes/No	Yes/No	Yes/No
Floating-point specifiers <code>a</code> , and <code>A</code>	No	No	Yes
Floating-point specifiers <code>e</code> , <code>E</code> , <code>f</code> , <code>F</code> , <code>g</code> , and <code>G</code>	No	No	Yes
Conversion specifier <code>n</code>	No	No	Yes
Scan set <code>[</code> and <code>]</code>	No	Yes	Yes
Assignment suppressing <code>*</code>	No	Yes	Yes
<code>long</code> <code>long</code> support	No	No	Yes

Table 7: Formatters for `scanf`

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 115.



Specifying the scanf formatter in the IDE

To explicitly specify a formatter, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



Specifying the scanf formatter from the command line

To explicitly specify a formatter, use one of these ILINK command line options:

```
--redirect _Scanf=_ScanfFull
--redirect _Scanf=_ScanfFullNoMb
--redirect _Scanf=_ScanfLarge
--redirect _Scanf=_ScanfLargeNoMb
--redirect _Scanf=_ScanfSmall
--redirect _Scanf=_ScanfSmallNoMb
```

Application debug support

In addition to the tools that generate debug information, there is a debug version of the library low-level interface (typically, I/O handling and basic runtime support). Using the debug library, your application can perform things like opening a file on the host computer and redirecting `stdout` to the debugger Terminal I/O window.

INCLUDING C-SPY DEBUGGING SUPPORT

You can make the library provide debugging support for:

- Handling program abort, exit, and assertions
- I/O handling, which means that `stdin` and `stdout` are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.

If you build your application project with the ILINK option **Semihosted** (`--semihosting`) or **IAR breakpoint** (`--semihosting=iar_breakpoint`), certain functions in the library are replaced by functions that communicate with the debugger.



To set linker options for debug support in the IDE, choose **Project>Options>General Options**. On the **Library configuration** page, select the **Semihosted** option or the **IAR breakpoint** option.

Note that for some Cortex-M devices it is also possible to direct `stdout/stderr` via SWO. This can significantly improve `stdout/stderr` performance compared to semihosting. For hardware requirement, see the *C-SPY® Debugging Guide for ARM®*.



To enable `stdout` via SWO on the command line, use the linker option `--redirect __iar_sh_stdout=__iar_sh_stdout_swo`.



To enable `stdout` via SWO in the IDE, choose **Project>Options>General Options**. On the **Library configuration** page, select the **Semihosted** option and the **stdout/stderr via SWO** option.

THE DEBUG LIBRARY FUNCTIONALITY

The debug library is used for communication between the application being debugged and the debugger itself. The debugger provides runtime services to the application via the low-level DLIB interface; services that allow capabilities like file and terminal I/O to be performed on the host computer.

These capabilities can be valuable during the early development of an application, for example in an application that uses file I/O before any flash file system I/O drivers are implemented. Or, if you need to debug constructions in your application that use `stdin` and `stdout` without the actual hardware device for input and output being available. Another use is producing debug printouts.

The low-level debugger runtime interface provided by DLIB is compatible with the semihosting interface provided by ARM Limited. When an application invokes a semihosting call, the execution stops at a debugger breakpoint. The debugger then handles the call, performs any necessary actions on the host computer and then resumes the execution.

The semihosting mechanism

There are three variants of semihosting mechanisms available:

- For Cortex-M, the interface uses `BKPT` instructions to perform semihosting calls
- For other ARM cores, `SVC` instructions are used for the semihosting calls
- *IAR breakpoint*, which is an IAR-specific alternative to semihosting that uses `SVC`.

To support semihosting via `SVC`, the debugger must set its semihosting breakpoint on the Supervisor Call vector to catch `SVC` calls. If your application uses `SVC` calls for other purposes than semihosting, the handling of this breakpoint will cause a severe performance penalty for each such call. *IAR breakpoint* is a way to get around this. By using a special function call instead of an `SVC` instruction to perform semihosting, the semihosting breakpoint can be set on that special function instead. This means that semihosting will not interfere with other uses of the Supervisor Call vector.

Note that *IAR breakpoint* is an IAR-specific extension of the semihosting standard. If you link your application with libraries built with toolchains from other vendors than IAR Systems and use *IAR breakpoint*, semihosting calls made from code in those libraries will not work.

THE C-SPY TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

Note: The Terminal I/O window is not opened automatically just because `__read` or `__write` is called; you must open it manually.

For more information about the Terminal I/O window, see the *C-SPY® Debugging Guide for ARM®*.

Speeding up terminal output

On some systems, terminal output might be slow because the host computer and the target hardware must communicate for each character.

For this reason, a replacement for the `__write` function called `__write_buffered` is included in the DLIB library. This module buffers the output and sends it to the debugger one line at a time, speeding up the output. Note that this function uses about 80 bytes of RAM memory.

To use this feature you can either choose **Project>Options>General Options>Library Options** and select the option **Buffered terminal output** in the IDE, or add this to the linker command line:

```
--redirect __write=__write_buffered
```

LOW-LEVEL FUNCTIONS IN THE DEBUG LIBRARY

The debug library contains implementations of the following low-level functions:

Function in DLIB low-level interface	Action
abort	Exits the application
clock	Returns the clock on the host computer
__close	Closes the associated host file on the host computer
__exit	Notifies that the end of the application was reached
__open	Opens a file on the host computer
__read	Directs <code>stdin</code> to the Terminal I/O window; all other files will read the associated host file
remove	Removes a file on the host computer
rename	Renames a file on the host computer

Table 8: Functions with special meanings when linked with debug library

Function in DLIB low-level interface	Action
<code>__iar_ReportAssert</code>	Prints an assert message to terminal I/O
<code>__lseek</code>	Searches in the associated host file on the host computer
<code>__write</code>	Directs <code>stdout</code> and <code>stderr</code> to the Terminal I/O window, all other files will write to the associated host file

Table 8: Functions with special meanings when linked with debug library (Continued)

Note: You should not use these low-level functions in your application. Instead you should use the high-level functions that use these functions to perform their actions. For more information, see *Library low-level interface, page 104*.

Adapting the library for target hardware

The library uses a set of low-level functions for handling accesses to your target system. To make these accesses work, you must implement your own version of these functions. These low-level functions are referred to as the *library low-level interface*.

When you have implemented your low-level interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules, page 105*.

LIBRARY LOW-LEVEL INTERFACE

The library uses a set of low-level functions to communicate with the target system. For example, `printf` and all other standard output functions use the low-level function `__write` to send the actual characters to an output device. Most of the low-level functions, like `__write`, have no implementation. Instead, you must implement them yourself to match your hardware.

However, the library contains a debug version of the library low-level interface, where the low-level functions are implemented so that they interact with the host computer via the debugger, instead of with the target hardware. If you use the debug library, your application can perform tasks like writing to the Terminal I/O window, accessing files on the host computer, getting the time from the host computer, etc. For more information, see *The debug library functionality, page 102*.

Note that your application should not use the low-level functions directly. Instead you should use the corresponding standard library function. For example, to write to `stdout`, you should use standard library functions like `printf` or `puts`, instead of `__write`.

The library files that you can override with your own versions are located in the `arm\src\lib` directory.

The low-level interface is further described in these sections:

- *Standard streams for input and output, page 113*
- *File input and output, page 117*
- *Signal and raise, page 121*
- *Time, page 121*
- *Assert, page 122.*

Overriding library modules

To use a library low-level interface that you have implemented, add it to your application. See *Adapting the library for target hardware, page 104*. Or, you might want to override a default library routine with your customized version. In both cases, follow this procedure:

- 1** Use a template source file—a library source file or another template—and copy it to your project directory.
- 2** Modify the file.
- 3** Add the customized file to your project, like any other source file.

Note: If you have implemented a library low-level interface and added it to a project that you have built with debug support, your low-level functions will be used and not the C-SPY debug support modules. For example, if you replace the debug support module `__write` with your own version, the C-SPY Terminal I/O window will not be supported.

The library files that you can override with your own versions are located in the `arm\src\lib` directory.

Building and using a customized library

Building a customized library is a complex process. Therefore, consider carefully whether it is really necessary. You must build your own C/C++ standard library when you want to define your own library configuration with support for locale, file descriptors, multibyte characters, etc.

In those cases, you must:

- Set up a library project

- Make the required library modifications
- Build your customized library
- Finally, make sure your application project will use the customized library.

Note: To build IAR Embedded Workbench projects from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`). However, no make or batch files for building the library from the command line are provided.

For information about the build process and the IAR Command Line Build Utility, see the *IDE Project Management and Building Guide for ARM®*.

SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template uses the Full library configuration, see Table 9, *Library configurations*, page 112.



In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 48.

Note: There is one important restriction on setting options. If you set an option on file level (file level override), no options on higher levels that operate on files will affect that file.

MODIFYING THE LIBRARY FUNCTIONALITY

You must modify the library configuration file and build your own library if you want to modify support for, for example, locale, file descriptors, and multibyte characters. This will include or exclude certain parts of the runtime environment.

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file, which sets up that specific library with the required library configuration. For more information, see Table 5, *Customizable items*, page 98.

The library configuration file is used for tailoring a build of the runtime library, and for tailoring the system header files.

Modifying the library configuration file

In your library project, open the file library configuration file and customize it by setting the values of the configuration symbols according to the application requirements.

When you are finished, build your library project with the appropriate project options.

USING A CUSTOMIZED LIBRARY

After you build your library, you must make sure to use it in your application project.



In the IDE you must do these steps:

- 1 Choose **Project>Options** and click the **Library Configuration** tab in the **General Options** category.
- 2 Choose **Custom DLIB** from the **Library** drop-down menu.
- 3 In the **Configuration file** text box, locate your library configuration file.
- 4 Click the **Library** tab, also in the **Linker** category. Use the **Additional libraries** text box to locate your library file.

System startup and termination

This section describes the runtime environment actions performed during startup and termination of your application.

The code for handling startup and termination is located in the source files `cstartup.s`, `cmain.s`, `cexit.s`, and `low_level_init.c` or `low_level_init.s` located in the `arm\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 111.

SYSTEM STARTUP

During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

For the hardware initialization, it looks like this:

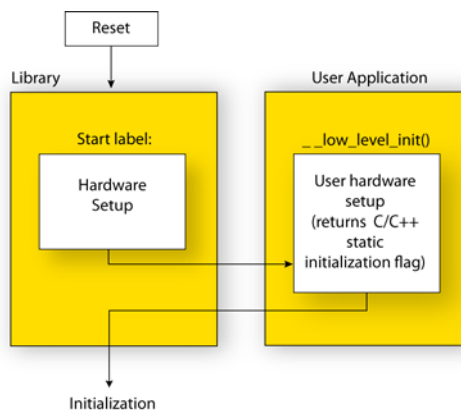


Figure 10: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__iar_program_start` in the system startup code.
- Exception stack pointers are initialized to the end of each corresponding section
- The stack pointer is initialized to the end of the `CSTACK` block
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

Note: For Cortex-M devices, the second bullet in the above list does not apply. The first and the third bullets are handled slightly differently. At reset, a Cortex-M CPU initializes PC and SP from the vector table (`__vector_table`), which is defined in the `cstartup_M.c` file.

For the C/C++ initialization, it looks like this:

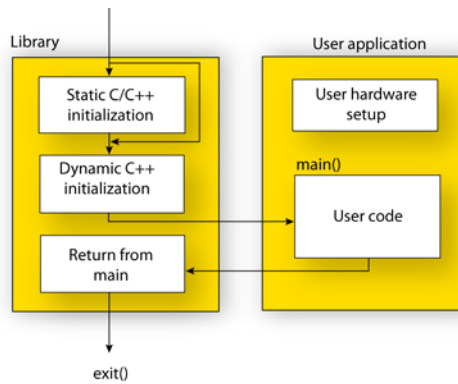


Figure 11: C/C++ initialization phase

- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialization at system startup*, page 74
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

For information about the initialization phase, see *Application execution—an overview*, page 42.

SYSTEM TERMINATION

This illustration shows the different ways an embedded application can terminate in a controlled way:

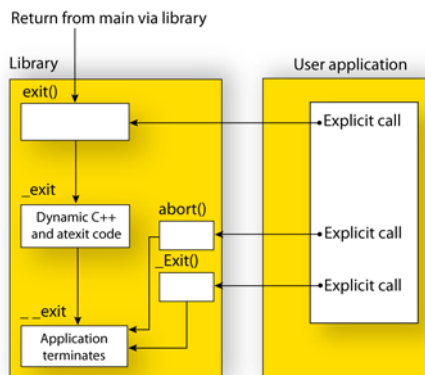


Figure 12: System termination phase

An application can terminate normally in two different ways:

- Return from the `main` function
- Call the `exit` function.

Because the C standard states that the two methods should be equivalent, the system startup code calls the `exit` function if `main` returns. The parameter passed to the `exit` function is the return value of `main`.

The default `exit` function is written in C. It calls a small assembler function `_exit` that will perform these operations:

- Call functions registered to be executed when the application ends. This includes C++ destructors for static and global variables, and functions registered with the standard function `atexit`
- Close all open files
- Call `__exit`
- When `__exit` is reached, stop the system.

An application can also exit by calling the `abort` or the `_Exit` function. The `abort` function just calls `__exit` to halt the system, and does not perform any type of cleanup. The `_Exit` function is equivalent to the `abort` function, except for the fact that `_Exit` takes an argument for passing exit status information.

If you want your application to do anything extra at exit, for example resetting the system, you can write your own implementation of the `__exit(int)` function.

C-SPY interface to system termination

If your project is linked with the semihosted interface, the normal `__exit` function is replaced with a special one. C-SPY will then recognize when this function is called and can take appropriate actions to simulate program termination. For more information, see *Application debug support*, page 101.

Customizing system initialization

It is likely that you need to customize the code for system initialization. For example, your application might need to initialize memory-mapped special function registers (SFRs), or omit the default initialization of data sections performed by `cstartup`.

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cmain.s` before the data sections are initialized. Modifying the file `cstartup.s` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s` and `low_level_init.c`, located in the `arm\src\lib` directory.

Note: Normally, you do not need to customize either of the files `cmain.s` or `cexit.s`.



If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 105.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s`, you do not have to rebuild the library.

__LOW_LEVEL_INIT

Two skeleton low-level initialization files are supplied with the product: a C source file, `low_level_init.c` and an alternative assembler source file, `low_level_init.s`. The latter is part of the prebuilt runtime environment. The only limitation using the C source version is that static initialized variables cannot be used within the file, as variable initialization has not been performed at this point.

The value returned by `__low_level_init` determines whether or not data sections should be initialized by the system startup code. If the function returns 0, the data sections will not be initialized.

MODIFYING THE FILE CSTARTUP.S

As noted earlier, you should not modify the file `cstartup.s` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s`, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see *Overriding library modules*, page 105.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s`. For information about how to change the start label used by the linker, see *--entry*, page 270.

For Cortex-M, you must create a modified copy of `cstartup_M.s` or `cstartup_M.c` to use interrupts or other exception handlers.

Library configurations

It is possible to configure the level of support for, for example, locale, file descriptors, multibyte characters.

The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The configuration file is used for tailoring a build of a runtime library, and tailoring the system header files used when compiling your application. The less functionality you need in the runtime environment, the smaller it becomes.

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities.

These predefined library configurations are available:

Library configuration	Description
Normal DLIB (default)	No locale interface, C locale, no file descriptor support, and no multibyte characters in <code>printf</code> and <code>scanf</code> .
Full DLIB	Full locale interface, C locale, file descriptor support, and multibyte characters in <code>printf</code> and <code>scanf</code> .

Table 9: Library configurations

CHOOSING A RUNTIME CONFIGURATION

To choose a runtime configuration, use one of these methods:

- Default prebuilt configuration—if you do not specify a library configuration explicitly you will get the default configuration. A configuration file that matches the runtime library object file will automatically be used.
- Prebuilt configuration of your choice—to specify a runtime configuration explicitly, use the `--dlib_config` compiler option. See *--dlib_config*, page 234.
- Your own configuration—you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For more information, see *Building and using a customized library*, page 105.

The prebuilt libraries are based on the default configurations, see Table 9, *Library configurations*.

Standard streams for input and output

Standard communication channels (streams) are defined in `stdio.h`. If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you must customize the low-level functionality to suit your hardware.

There are low-level I/O functions, which are the fundamental functions through which C and C++ performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these functions using whatever facilities the hardware environment provides. For more information about implementing low-level functions, see *Adapting the library for target hardware*, page 104.

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `arm\src\lib` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 105. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *Application debug support*, page 101.

Example of using `__write`

The code in this example uses memory-mapped I/O to write to an LCD display, whose port is assumed to be located at address 0x1000:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 0x1000;

size_t __write(int handle,
               const unsigned char *buf,
               size_t bufSize)

{
    size_t nChars = 0;

    /* Check for the command to flush all handles */
    if (handle == -1)
    {
        return 0;
    }

    /* Check for stdout and stderr
       (only necessary if FILE descriptors are enabled.) */
    if (handle != 1 && handle != 2)
    {
        return -1;
    }

    for (/* Empty */; bufSize > 0; --bufSize)
    {
        lcdIO = *buf;
        ++buf;
        ++nChars;
    }

    return nChars;
}
```

Note: When DLIB calls `__write`, DLIB assumes the following interface: a call to `__write` where `buf` has the value `NULL` is a command to flush the stream. When the `handle` is `-1`, all streams should be flushed.

Example of using `__read`

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at 0x1000:

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0x1000;

size_t __read(int handle,
              unsigned char *buf,
              size_t bufSize)
{
    size_t nChars = 0;

    /* Check for stdin
       (only necessary if FILE descriptors are enabled) */
    if (handle != 0)
    {
        return -1;
    }

    for (/*Empty*/; bufSize > 0; --bufSize)
    {
        unsigned char c = kbIO;
        if (c == 0)
            break;

        *buf++ = c;
        ++nChars;
    }

    return nChars;
}
```

For information about the @ operator, see *Controlling data and function placement in memory*, page 191.

Configuration symbols for `printf` and `scanf`

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Choosing formatters for `printf` and `scanf`*, page 99.

If the provided formatters do not meet your requirements, you can customize the full formatters. However, that means you must rebuild the runtime library.

The default behavior of the `printf` and `scanf` formatters are defined by configuration symbols in the file `DLib_Defaults.h`.

These configuration symbols determine what capabilities the function `printf` should have:

Printf configuration symbols	Includes support for
<code>_DLIB_PRINTF_MULTIBYTE</code>	Multibyte characters
<code>_DLIB_PRINTF_LONG_LONG</code>	Long long (ll qualifier)
<code>_DLIB_PRINTF_SPECIFIER_FLOAT</code>	Floating-point numbers
<code>_DLIB_PRINTF_SPECIFIER_A</code>	Hexadecimal floating-point numbers
<code>_DLIB_PRINTF_SPECIFIER_N</code>	Output count (%n)
<code>_DLIB_PRINTF_QUALIFIERS</code>	Qualifiers h, l, L, v, t, and z
<code>_DLIB_PRINTF_FLAGS</code>	Flags -, +, #, and 0
<code>_DLIB_PRINTF_WIDTH_AND_PRECISION</code>	Width and precision
<code>_DLIB_PRINTF_CHAR_BY_CHAR</code>	Output char by char or buffered

Table 10: Descriptions of printf configuration symbols

When you build a library, these configurations determine what capabilities the function `scanf` should have:

Scanf configuration symbols	Includes support for
<code>_DLIB_SCANF_MULTIBYTE</code>	Multibyte characters
<code>_DLIB_SCANF_LONG_LONG</code>	Long long (ll qualifier)
<code>_DLIB_SCANF_SPECIFIER_FLOAT</code>	Floating-point numbers
<code>_DLIB_SCANF_SPECIFIER_N</code>	Output count (%n)
<code>_DLIB_SCANF_QUALIFIERS</code>	Qualifiers h, j, l, t, z, and L
<code>_DLIB_SCANF_SCANSET</code>	Scanset ([*])
<code>_DLIB_SCANF_WIDTH</code>	Width
<code>_DLIB_SCANF_ASSIGNMENT_SUPPRESSING</code>	Assignment suppressing ([*])

Table 11: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must;

- 1 Set up a library project, see *Building and using a customized library*, page 105.
- 2 Define the configuration symbols according to your application requirements.

File input and output

The library contains a large number of powerful functions for file I/O operations, such as `fopen`, `fclose`, `fprintf`, `fputs`, etc. All these functions call a small set of low-level functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters. Before your application can use the library functions for file I/O operations, you must implement the corresponding low-level function to suit your target hardware. For more information, see *Adapting the library for target hardware*, page 104.

Note that file I/O capability in the library is only supported by libraries with the full library configuration, see *Library configurations*, page 112. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

Template code for these I/O files is included in the product:

I/O function	File	Description
<code>__close</code>	<code>close.c</code>	Closes a file.
<code>__lseek</code>	<code>lseek.c</code>	Sets the file position indicator.
<code>__open</code>	<code>open.c</code>	Opens a file.
<code>__read</code>	<code>read.c</code>	Reads a character buffer.
<code>__write</code>	<code>write.c</code>	Writes a character buffer.
<code>remove</code>	<code>remove.c</code>	Removes a file.
<code>rename</code>	<code>rename.c</code>	Renames a file.

Table 12: Low-level I/O files

The low-level functions identify I/O streams, such as an open file, with a file descriptor that is a unique integer. The I/O streams normally associated with `stdin`, `stdout`, and `stderr` have the file descriptors 0, 1, and 2, respectively.

Note: If you link your application with I/O debug support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Application debug support*, page 101.

Locale

Locale is a part of the C language that allows language- and country-specific settings for several areas, such as currency symbols, date and time, and multibyte character encoding.

Depending on what runtime library you are using you get different level of locale support. However, the more locale support, the larger your code will get. It is therefore necessary to consider what level of support your application needs.

The DLIB library can be used in two main modes:

- With locale interface, which makes it possible to switch between different locales during runtime
- Without locale interface, where one selected locale is hardwired into the application.

LOCALE SUPPORT IN PREBUILT LIBRARIES

The level of locale support in the prebuilt libraries depends on the library configuration.

- All prebuilt libraries support the C locale only
- All libraries with *full library configuration* have support for the locale interface. For prebuilt libraries with locale interface, it is by default only supported to switch multibyte character encoding at runtime.
- Libraries with *normal library configuration* do not have support for the locale interface.

If your application requires a different locale support, you must rebuild the library.

CUSTOMIZING THE LOCALE SUPPORT

If you decide to rebuild the library, you can choose between these locales:

- The Standard C locale
- The POSIX locale
- A wide range of European locales.

Locale configuration symbols

The configuration symbol `_DLIB_FULL_LOCALE_SUPPORT`, which is defined in the library configuration file, determines whether a library has support for a locale interface or not. The locale configuration symbols `_LOCALE_USE_LANG_REGION` and `_ENCODING_USE_ENCODING` define all the supported locales and encodings:

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C           /* C locale */
#define _LOCALE_USE_EN_US      /* American English */
#define _LOCALE_USE_EN_GB      /* British English */
#define _LOCALE_USE_SV_SE      /* Swedish in Sweden */
```

See `DLib_Defaults.h` for a list of supported locale and encoding settings.

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 105.

Note: If you use multibyte characters in your C or assembler source code, make sure that you select the correct locale symbol (the local host locale).

Building a library without support for locale interface

The locale interface is not included if the configuration symbol `_DLIB_FULL_LOCALE_SUPPORT` is set to 0 (zero). This means that a hardwired locale is used—by default the Standard C locale—but you can choose one of the supported locale configuration symbols. The `setlocale` function is not available and can therefore not be used for changing locales at runtime.

Building a library with support for locale interface

Support for the locale interface is obtained if the configuration symbol `_DLIB_FULL_LOCALE_SUPPORT` is set to 1. By default, the Standard C locale is used, but you can define as many configuration symbols as required. Because the `setlocale` function will be available in your application, it will be possible to switch locales at runtime.

CHANGING LOCALES AT RUNTIME

The standard library function `setlocale` is used for selecting the appropriate portion of the application's locale when the application is running.

The `setlocale` function takes two arguments. The first one is a locale category that is constructed after the pattern `LC_CATEGORY`. The second argument is a string that describes the locale. It can either be a string previously returned by `setlocale`, or it can be a string constructed after the pattern:

`lang_REGION`

or

`lang_REGION.encoding`

The `lang` part specifies the language code, and the `REGION` part specifies a region qualifier, and `encoding` specifies the multibyte character encoding that should be used.

The `lang_REGION` part matches the `_LOCALE_USE_LANG_REGION` preprocessor symbols that can be specified in the library configuration file.

Example

This example sets the locale configuration symbols to Swedish to be used in Finland and UTF8 multibyte character encoding:

```
setlocale (LC_ALL, "sv_FI.Utf8");
```

Environment interaction

According to the C standard, your application can interact with the environment using the functions `getenv` and `system`.

Note: The `putenv` function is not required by the standard, and the library does not provide an implementation of it.

THE GETENV FUNCTION

The `getenv` function searches the string, pointed to by the global variable `__environ`, for the key that was passed as argument. If the key is found, the value of it is returned, otherwise 0 (zero) is returned. By default, the string is empty.

To create or edit keys in the string, you must create a sequence of null terminated strings where each string has the format:

```
key=value\0
```

End the string with an extra null character (if you use a C string, this is added automatically). Assign the created sequence of strings to the `__environ` variable.

For example:

```
const char MyEnv[] = "Key=Value\0Key2=Value2\0";
__environ = MyEnv;
```

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `arm\src\lib` directory. For information about overriding default library modules, see *Overriding library modules, page 105*.

THE SYSTEM FUNCTION

If you need to use the `system` function, you must implement it yourself. The `system` function available in the library simply returns -1.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library, page 105*.

Note: If you link your application with support for I/O debugging, the `system` function is replaced by a C-SPY variant. For more information, see *Application debug support*, page 101.

Signal and raise

Default implementations of the functions `signal` and `raise` are available. If these functions do not provide the functionality that you need, you can implement your own versions.

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `arm\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 105.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 105.

Time

To make the `__time32`, `__time64`, and `date` functions work, you must implement the functions `clock`, `__time32`, `__time64`, and `__getzone`. Whether you use `__time32` or `__time64` depends on which interface you use for `time_t`, see *time.h*, page 380.

To implement these functions does not require that you rebuild the library. You can find source templates in the files `clock.c`, `time.c`, `time64.c`, and `getzone.c` in the `arm\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 105.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 105.

The default implementation of `__getzone` specifies UTC (Coordinated Universal Time) as the time zone.

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For more information, see *Application debug support*, page 101.

Pow

The DLIB runtime library contains an alternative power function with extended precision, `powXp`. The lower-precision `pow` function is used by default. To use the `powXp` function instead, link your application with the linker option `--redirect pow=powXp`.

Assert

If you linked your application with support for runtime debugging, an assert will print a message on `stdout`. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. The `__iar_ReportAssert` function generates the assert notification. You can find template code in the `arm\src\lib` directory. For more information, see *Building and using a customized library*, page 105. To turn off assertions, you must define the symbol `NDEBUG`.



In the IDE, this symbol `NDEBUG` is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See *NDEBUG*, page 371.

Atexit

The linker allocates a static memory area for `atexit` function calls. By default, the number of calls to the `atexit` function are limited to 32. To change this limit, see *Setting up the atexit limit*, page 86.

Managing a multithreaded environment

In a multithreaded environment, the standard library must treat all library objects according to whether they are global or local to a thread. If an object is a true global object, any updates of its state must be guarded by a locking mechanism to make sure that only one thread can update it at any given time. If an object is local to a thread, the static variables containing the object state must reside in a variable area local to that thread. This area is commonly named *thread-local storage* (TLS).

There are three possible scenarios, and you need to consider which one that applies to you:

- If you are using an RTOS that supports the multithreading provided by the DLIB library, the RTOS and the DLIB library will handle multithreading which means you do not need to adapt the DLIB library.
- If you are using an RTOS that does not support or only partly supports the multithreading provided by the DLIB library, you probably need to adapt both the RTOS and the DLIB library.
- If you are not using an RTOS, you must adapt the DLIB library to get support for multithreading.

MULTITHREAD SUPPORT IN THE DLIB LIBRARY

The DLIB library uses two kinds of locks—*system locks* and *file stream locks*. The file stream locks are used as guards when the state of a file stream is updated, and are only needed in the Full library configuration. The following library objects are guarded with system locks:

- The heap, in other words when `malloc`, `new`, `free`, `delete`, `realloc`, or `calloc` is used.
- The file system (only available in the Full library configuration), but not the file streams themselves. The file system is updated when a stream is opened or closed, in other words when `fopen`, `fclose`, `fdopen`, `fflush`, or `freopen` is used.
- The signal system, in other words when `signal` is used.
- The temporary file system, in other words when `tmpnam` is used.
- Initialization of static function objects.

These library objects use TLS:

Library objects using TLS	When these functions are used
Error functions	<code>errno</code> , <code>strerror</code>
Locale functions	<code>localeconv</code> , <code>setlocale</code>
Time functions	<code>asctime</code> , <code>localtime</code> , <code>gmtime</code> , <code>mktime</code>
Multibyte functions	<code>mbrlen</code> , <code>mbrtowc</code> , <code>mbsrtowc</code> , <code>mbtowc</code> , <code>wcrtomb</code> , <code>wcsrtomb</code> , <code>wctomb</code>
Rand functions	<code>rand</code> , <code>srand</code>
Miscellaneous functions	<code>atexit</code> , <code>strtok</code>
C++ exception engine	N/A

Table 13: Library objects using TLS

ENABLING MULTITHREAD SUPPORT

To enable multithread support in the library, you must:

- Implement code for the library’s system locks interface
- If file streams are used, implement code for the library’s file stream locks interface or redirect the interface to the system locks interface (using the linker option `--redirect`)
- Implement source code that handles thread creation, thread destruction, and TLS access methods for the library
- Modify the linker configuration file accordingly

- If any of the C++ variants are used, use the compiler option `--guard_calls`. Otherwise, function static variables with dynamic initializers might be initialized simultaneously by several threads.

You can find the required declaration of functions and definitions of macros in the `DLib_Threads.h` file, which is included by `yvals.h`.

System locks interface

This interface must be fully implemented for system locks to work:

```
typedef void *__iar_Rmtx;                /* Lock info object */

void __iar_system_Mtxinit(__iar_Rmtx *); /* Initialize a system
                                           lock */
void __iar_system_Mtxdst(__iar_Rmtx *); /* Destroy a system lock */
void __iar_system_Mtxlock(__iar_Rmtx *); /* Lock a system lock */
void __iar_system_Mtxunlock(__iar_Rmtx *); /* Unlock a system
                                           lock */
```

The lock and unlock implementation must survive nested calls.

File streams locks interface

This interface is only needed for the Full library configuration. If file streams are used, they can either be fully implemented or they can be redirected to the system locks interface. This interface must be implemented for file streams locks to work:

```
typedef void *__iar_Rmtx;                /* Lock info object */

void __iar_file_Mtxinit(__iar_Rmtx *); /* Initialize a file lock */
void __iar_file_Mtxdst(__iar_Rmtx *); /* Destroy a file lock */
void __iar_file_Mtxlock(__iar_Rmtx *); /* Lock a file lock */
void __iar_file_Mtxunlock(__iar_Rmtx *); /* Unlock a file lock */
```

The lock and unlock implementation must survive nested calls.

DLIB lock usage

The number of locks that the DLIB library assumes exist are:

- `_FOPEN_MAX`—the maximum number of file stream locks. These locks are only used in the Full library configuration, in other words only if both the macro symbols `_DLIB_FILE_DESCRIPTOR` and `_FILE_OP_LOCKS` are true.
- `_MAX_LOCK`—the maximum number of system locks.

Note that even if the application uses fewer locks, the DLIB library will initialize and destroy all of the locks above.

For information about the initialization and destruction code, see `xsyslock.c`.

TLS handling

The DLIB library supports TLS memory areas for two types of threads: the *main thread* (the `main` function including the system startup and exit code) and *secondary threads*.

The main thread’s TLS memory area:

- Is automatically created and initialized by your application’s startup sequence
- Is automatically destructed by the application’s destruct sequence
- Is located in the section `__DLIB_PERTHREAD`
- Exists also for non-threaded applications.

Each secondary thread’s TLS memory area:

- Must be manually created and initialized
- Must be manually destructed
- Is located in a manually allocated memory area.

If you need the runtime library to support secondary threads, you must override the function:

```
void *__iar_dlib_perthread_access(void *sympb);
```

The parameter is the address to the TLS variable to be accessed—in the main thread’s TLS area—and it should return the address to the symbol in the current TLS area.

Two interfaces can be used for creating and destroying secondary threads. You can use the following interface that allocates a memory area on the heap and initializes it. At deallocation, it destroys the objects in the area and then frees the memory.

```
void *__iar_dlib_perthread_allocate(void);
void __iar_dlib_perthread_deallocate(void *);
```

Alternatively, if the application handles the TLS allocation, you can use this interface for initializing and destroying the objects in the memory area:

```
void __iar_dlib_perthread_initialize(void *);
void __iar_dlib_perthread_destroy(void *);
```

These macros can be helpful when you implement an interface for creating and destroying secondary threads:

Macro	Description
<code>__IAR_DLIB_PERTHREAD_SIZE</code>	Returns the size needed for the TLS memory area.

Table 14: Macros for implementing TLS allocation

Macro	Description
<code>__IAR_DLIB_PERTHREAD_INIT_SIZE</code>	Returns the initializer size for the TLS memory area. You should initialize the rest of the TLS memory area, up to <code>__IAR_DLIB_PERTHREAD_SIZE</code> to zero.
<code>__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(<i>symbolptr</i>)</code>	Returns the offset to the symbol in the TLS memory area.

Table 14: Macros for implementing TLS allocation (Continued)

Note that the size needed for TLS variables depends on which DLIB resources your application uses.

This is an example of how you can handle threads:

```
#include <yvals.h>

/* A thread's TLS pointer */
void _DLIB_TLS_MEMORY *TLSp;

/* Are we in a secondary thread? */
int InSecondaryThread = 0;

/* Allocate a thread-local TLS memory
   area and store a pointer to it in TLSp. */
void AllocateTLS()
{
    TLSp = __iar_dlib_perthread_allocate();
}

/* Deallocate the thread-local TLS memory area. */
void DeallocateTLS()
{
    __iar_dlib_perthread_deallocate(TLSp);
}

/* Access an object in the
   thread-local TLS memory area. */
void _DLIB_TLS_MEMORY *__iar_dlib_perthread_access(
    void _DLIB_TLS_MEMORY *sympb)
{
    char _DLIB_TLS_MEMORY *p = 0;
    if (InSecondaryThread)
        p = (char _DLIB_TLS_MEMORY *) TLSp;
    else
        p = (char _DLIB_TLS_MEMORY *)
            __segment_begin("__DLIB_PERTHREAD");
}
```

```

p += __IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(sympb) ;
return (void _DLIB_TLS_MEMORY *) p;
}

```

The `TLSp` variable is unique for each thread, and must be exchanged by the RTOS or manually whenever a thread switch occurs.

CHANGES IN THE LINKER CONFIGURATION FILE

Normally, the linker automatic chooses how to initialize static data. If threads are used, the main thread's TLS memory area must be initialized by naive copying because the initializers are used for each secondary thread's TLS memory area as well. Insert the following statement in your linker configuration file:

```

initialize by copy with packing = none { section __DLIB_PERTHREAD
};

```

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the tools provided by IAR Systems to ensure that modules that are linked into an application are compatible, in other words, are built using compatible settings. The tools use a set of predefined runtime model attributes. In addition to these, you can define your own that you can use to ensure that incompatible modules are not used together.

Note: In addition to the predefined attributes, compatibility is also checked against the AEABI runtime attributes. These attributes deal mainly with object code compatibility, etc. They reflect compilation settings and are not user-configurable.

RUNTIME MODEL ATTRIBUTES

A runtime attribute is a pair constituted of a named key and its corresponding value. In general, two modules can only be linked together if they have the same value for each key that they both define.

There is one exception: if the value of an attribute is `*`, then that attribute matches any value. The reason for this is that you can specify this in a module to show that you have considered a consistency property, and this ensures that the module does not rely on that property.

Note: For IAR predefined runtime model attributes, the linker checks them in several ways.

Example

In this table, the object files could (but do not have to) define the two runtime attributes `color` and `taste`:

Object file	Color	Taste
file1	blue	not defined
file2	red	not defined
file3	red	*
file4	red	spicy
file5	red	lean

Table 15: Example of runtime model attributes

In this case, `file1` cannot be linked with any of the other files, since the runtime attribute `color` does not match. Also, `file4` and `file5` cannot be linked together, because the `taste` runtime attribute does not match.

On the other hand, `file2` and `file3` can be linked with each other, and with either `file4` or `file5`, but not with both.

USING RUNTIME MODEL ATTRIBUTES

To ensure module consistency with other object files, use the `#pragma rtmodel` directive to specify runtime model attributes in your C/C++ source code. For example, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example `uart`. For each mode, specify a value, for example `model` and `mode2`. Declare this in each module that assumes that the UART is in a particular mode. This is how it could look like in one of the modules:

```
#pragma rtmodel="uart", "model"
```

Alternatively, you can also use the `rtmodel` assembler directive to specify runtime model attributes in your assembler source code. For example:

```
rtmodel "uart", "model"
```

Note that key names that start with two underscores are reserved by the compiler. For more information about the syntax, see *rtmodel*, page 330 and the *ARM® IAR Assembler Reference Guide*, respectively,

At link time, the IAR ILINK Linker checks module consistency by ensuring that modules with conflicting runtime attributes will not be used together. If conflicts are detected, an error is issued.

Assembler language interface

When you develop an application for an embedded system, there might be situations where you will find it necessary to write parts of the code in assembler, for example when using mechanisms in the ARM core that require precise timing and special instruction sequences.

This chapter describes the available methods for this and some C alternatives, with their advantages and disadvantages. It also describes how to write functions in assembler language that work together with an application written in C or C++.

Finally, the chapter covers how functions are called, and how you can implement support for call frame information in your assembler routines for use in the C-SPY® Call Stack window.

Mixing C and assembler

The IAR C/C++ Compiler for ARM provides several ways to access low-level resources:

- Modules written entirely in assembler
- Intrinsic functions (the C alternative)
- Inline assembler.

It might be tempting to use simple inline assembler. However, you should carefully choose which method to use.

INTRINSIC FUNCTIONS

The compiler provides a few predefined functions that allow direct access to low-level processor operations without having to use the assembler language. These functions are known as intrinsic functions. They can be very useful in, for example, time-critical routines.

An intrinsic function looks like a normal function call, but it is really a built-in function that the compiler recognizes. The intrinsic functions compile into inline code, either as a single instruction, or as a short sequence of instructions.

The advantage of an intrinsic function compared to using inline assembler is that the compiler has all necessary information to interface the sequence properly with register allocation and variables. The compiler also knows how to optimize functions with such sequences; something the compiler is unable to do with inline assembler sequences. The result is that you get the desired sequence properly integrated in your code, and that the compiler can optimize the result.

For more information about the available intrinsic functions, see the chapter *Intrinsic functions*.

MIXING C AND ASSEMBLER MODULES

It is possible to write parts of your application in assembler and mix them with your C or C++ modules. This gives several benefits compared to using inline assembler:

- The function call mechanism is well-defined
- The code will be easy to read
- The optimizer can work with the C or C++ functions.

This causes some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all scratch registers are destroyed by an inline assembler instruction. In many cases, the overhead of the extra instructions can be removed by the optimizer.

An important advantage is that you will have a well-defined interface between what the compiler produces and what you write in assembler. When using inline assembler, you will not have any guarantees that your inline assembler lines do not interfere with the compiler generated code.

When an application is written partly in assembler language and partly in C or C++, you are faced with several questions:

- How should the assembler code be written so that it can be called from C?
- Where does the assembler code find its parameters, and how is the return value passed back to the caller?
- How should assembler code call functions written in C?
- How are global C variables accessed from code written in assembler language?
- Why does not the debugger display the call stack when assembler code is being debugged?

The first question is discussed in the section *Calling assembler routines from C*, page 136. The following two are covered in the section *Calling convention*, page 139.

The answer to the final question is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the *call frame*, which must be supplied as annotations in the assembler source file. For more information, see *Call frame information*, page 146.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 136, and *Calling assembler routines from C++*, page 138, respectively.

INLINE ASSEMBLER

Inline assembler can be used for inserting assembler instructions directly into a C or C++ function. Typically, this can be useful if you need to:

- Access hardware resources that are not accessible in C (in other words, when there is no definition for an SFR or there is no suitable intrinsic function available).
- Manually write a time-critical sequence of code that if written in C will not have the right timing.
- Manually write a speed-critical sequence of code that if written in C will be too slow.

An inline assembler statement is similar to a C function in that it can take input arguments (input operands), have return values (output operands), and read or write to C symbols (via the operands). An inline assembler statement can also declare *clobbered resources* (that is, values in registers and memory that have been overwritten).

Limitations

Most things you can do in normal assembler language are also possible with inline assembler, with the following differences:

- Alignment cannot be controlled; this means, for example, that `DC32` directives might be misaligned.
- The only accepted register synonyms are `SP` (for `R13`), `LR` (for `R14`), and `PC` (for `R15`).
- In general, assembler directives will cause errors or have no meaning. However, data definition directives will work as expected.
- Resources used (registers, memory, etc) that are also used by the C compiler must be declared as operands or clobbered resources.
- If you do not want to risk that the inline assembler statement is optimized away by the compiler, you must declare it `volatile`.

- Accessing a C symbol or using a constant expression requires the use of operands.
- Dependencies between the expressions for the operands might result in an error.

Risks with inline assembler

Without operands and clobbered resources, inline assembler statements have no interface with the surrounding C source code. This makes the inline assembler code fragile, and might also become a maintenance problem if you update the compiler in the future. There are also several limitations to using inline assembler without operands and clobbered resources:

- The compiler's various optimizations will disregard any effects of the inline statements, which will not be optimized at all.
- The inline assembler statement will be *volatile* and *clobbered memory* is implied. This means that the compiler will not remove the assembler statement. It will simply be inserted at the given location in the program flow. The consequences or side-effects that the insertion might have on the surrounding code are not taken into consideration. If, for example, registers or memory locations are altered, they might have to be restored within the sequence of inline assembler instructions for the rest of the code to work properly.



The following example (for ARM mode) demonstrates the risks of using the `asm` keyword without operands and clobbers:

```
int Add(int term1, int term2)
{
    asm("adds r0,r0,r1");
    return term1;
}
```

In this example:

- The function `Add` assumes that values are passed and returned in registers in a way that they might not always be, for example if the function is inlined.
- The `s` in the `adds` instruction implies that the condition flags are updated, which should be specified using the `cc` clobber operand. Otherwise, the compiler will assume that the condition flags are not modified.

Inline assembler without using operands or clobbered resources is therefore often best avoided.

Reference information for inline assembler

The `asm` and `__asm` keywords both insert inline assembler instructions. However, when you compile C source code, the `asm` keyword is not available when the option `--strict` is used. The `__asm` keyword is always available.

Syntax

The syntax of an inline assembler statement is (similar to the one used by GNU gcc):

```
asm [volatile] ( string [assembler-interface])
```

string can contain one or more valid assembler instructions or data definition assembler directives, separated by `\n`.

For example:

```
asm("label:nop\n"
    "b label");
```

Note that you can define and use local labels in inline assembler instructions.

assembler-interface is:

```
: comma-separated list of output operands      /* optional */
: comma-separated list of input operands       /* optional */
: comma-separated list of clobbered resources /* optional */
```

Operands

An inline assembler statement can have one input and one output comma-separated list of operands. Each operand consists of an optional symbolic name in brackets, a quoted constraint, followed by a C expression in parentheses.

Syntax of operands

```
[ [symbolic-name ] ] "[modifiers]constraint" (expr)
```

For example:

```
int Add(int term1, int term2)
{
    int sum;

    asm("add %0,%1,%2"
        : "=r"(sum)
        : "r" (term1), "r" (term2));

    return sum;
}
```

In this example, the assembler instruction uses one output operand, `sum`, two input operands, `term1` and `term2`, and no clobbered resources.

It is possible to omit any list by leaving it empty. For example:

```
int matrix[M][N];

void MatrixPreloadRow(int row)
{
    asm volatile ("pld [%0]" : : "r" (&matrix[row][0]));
}
```

Operand constraints

Constraint	Description
r	Uses a general purpose register for the expression: R0–R12, R14 (for ARM and Thumb2) R0–R7 (for Thumb1)
l	R0–R7 (only valid for Thumb1)

Table 16: Inline assembler operand constraints

Constraint modifiers

Constraint modifiers can be used together with a constraint to modify its meaning. This table lists the supported constraint modifiers:

Modifier	Description
=	Write-only operand
+	Read-write operand
&	Early clobber output operand which is written to before the instruction has processed all the input operands.

Table 17: Supported constraint modifiers

Referring to operands

Assembler instructions refer to operands by prefixing their order number with %. The first operand has order number 0 and is referred to by %0.

If the operand has a symbolic name, you can refer to it using the syntax %[operand.name]. Symbolic operand names are in a separate namespace from C/C++ code and can be the same as a C/C++ variable names. Each operand name must however be unique in each assembler statement. For example:

```
int Add(int term1, int term2)
{
    int sum;

    asm("add %[Rd], %[Rn], %[Rm] "
        : [Rd] "=r" (sum)
        : [Rn] "r" (term1), [Rm] "r" (term2));

    return sum;
}
```

Input operands

Input operands cannot have any modifiers, but they can have any valid C expression as long as the type of the expression fits the register.

The C expression will be evaluated just before any of the assembler instructions in the inline assembler statement and assigned to the constraint, for example a register.

Output operands

Output operands must have `=` as a modifier and the C expression must be an l-value and specify a writable location. For example, `=r` for a write-only general purpose register. The constraint will be assigned to the evaluated C expression (as an l-value) immediately after the last assembler instruction in the inline assembler statement. Input operands are assumed to be consumed before output is produced and the compiler may use the same register for an input and output operand. To prohibit this, prefix the output constraint with `&` to make it an early clobber resource, for example `=&r`. This will ensure that the output operand will be allocated in a different register than the input operands.

Input/output operands

An operand that should be used both for input and output must be listed as an output operand and have the `+` modifier. The C expression must be an l-value and specify a writable location. The location will be read immediately before any assembler instructions and it will be written to right after the last assembler instruction.

This is an example of using a read-write operand:

```
int Double(int value)
{
    asm("add %0,%0,%0" : "+r"(value));

    return value;
}
```

In the example above, the input value for `value` will be placed in a general purpose register. After the assembler statement, the result from the `ADD` instruction will be placed in the same register.

Clobbered resources

An inline assembler statement can have a list of clobbered resources.

```
"resource1", "resource2", ...
```

Specify clobbered resources to inform the compiler about which resources the inline assembler statement destroys. Any value that resides in a clobbered resource and that is needed after the inline assembler statement will be reloaded.

Clobbered resources will not be used as input or output operands.

This is an example of how to use clobbered resources:

```
int Add(int term1, int term2)
{
    int sum;

    asm("adds %0,%1,%2"
        : "=r"(sum)
        : "r" (term1), "r" (term2)
        : "cc");
}
```

```
    return sum;
}
```

In this example the condition codes will be modified by the `ADDS` instruction, therefore `"cc"` must be listed in the clobber list.

This table lists valid clobbered resources:

Clobber	Description
R0–R12, R14, for ARM mode and Thumb2 R0–R7, R12, R14 for Thumb1	General purpose registers
S0–S31, D0–D31, Q0–Q15	Floating-point registers
cc	The condition flags (N, Z, V, and C)
memory	To be used if the instructions modify any memory. This will avoid keeping memory values cached in registers across the inline assembler statement.

Table 18: List of valid clobbers

This is an example of how to use clobbered memory:

```
int StoreExclusive(unsigned long * location, unsigned long value)
{
    int failed;

    asm("strex %0,%2,[%1]"
        : "=&r" (failed)
        : "r" (location), "r" (value)
        : "memory");

    /* Note: 'strex' requires ARMv6 (ARM) or ARMv6T2 (THUMB) */

    return failed;
}
```

Calling assembler routines from C

An assembler routine that will be called from C must:

- Conform to the calling convention
- Have a `PUBLIC` entry-point label
- Be declared as external before any call, to allow type checking and optional promotion of parameters, as in these examples:

```
extern int foo(void);
```



```
or
extern int foo(int i, int j);
```

One way of fulfilling these requirements is to create skeleton code in C, compile it, and study the assembler list file.

CREATING SKELETON CODE

The recommended way to create an assembler language routine with the correct interface is to start with an assembler language source file created by the C compiler. Note that you must create skeleton code for each function prototype.

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the variables required and perform simple accesses to them. In this example, the assembler routine takes an `int` and a `char`, and then returns an `int`:

```
extern int gInt;
extern char gChar;

int Func(int arg1, char arg2)
{
    int locInt = arg1;
    gInt = arg1;
    gChar = arg2;
    return locInt;
}

int main()
{
    int locInt = gInt;
    gInt = Func(locInt, gChar);
    return 0;
}
```

Note: In this example we use a low optimization level when compiling the code to show local and global variable access. If a higher level of optimization is used, the required references to local variables could be removed during the optimization. The actual function declaration is not changed by the optimization level.

COMPILING THE CODE



In the IDE, specify list options on file level. Select the file in the workspace window. Then choose **Project>Options**. In the **C/C++ Compiler** category, select **Override inherited settings**. On the **List** page, deselect **Output list file**, and instead select the **Output assembler file** option and its suboption **Include source**. Also, be sure to specify a low level of optimization.



Use these options to compile the skeleton code:

```
icarm skeleton.c -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s`. Also remember to specify a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s`.

Note: The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file.



In the IDE, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**.



On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

- The calling convention
- The return values
- The global variables
- The function parameters
- How to create space on the stack (auto variables)
- Call frame information (CFI).

The CFI directives describe the call frame information needed by the Call Stack window in the debugger. For more information, see *Call frame information, page 146*.

Calling assembler routines from C++

The C calling convention does not apply to C++ functions. Most importantly, a function name is not sufficient to identify a C++ function. The scope and the type of the function are also required to guarantee type-safe linkage, and to resolve overloading.

Another difference is that non-static member functions get an extra, hidden argument, the `this` pointer.

However, when using C linkage, the calling convention conforms to the C calling convention. An assembler routine can therefore be called from C++ when declared in this manner:

```
extern "C"
{
    int MyRoutine(int);
}
```

The following example shows how to achieve the equivalent to a non-static member function, which means that the implicit `this` pointer must be made explicit. It is also possible to “wrap” the call to the assembler routine in a member function. Use an inline member function to remove the overhead of the extra call—this assumes that function inlining is enabled:

```
class MyClass;

extern "C"
{
    void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
public:
    inline void DoIt(int arg)
    {
        ::DoIt(this, arg);
    }
};
```

Calling convention

A calling convention is the way a function in a program calls another function. The compiler handles this automatically, but, if a function is written in assembler language, you must know where and how its parameters can be found, how to return to the program location from where it was called, and how to return the resulting value.

It is also important to know which registers an assembler-level routine must preserve. If the program preserves too many registers, the program might be ineffective. If it preserves too few registers, the result would be an incorrect program.

This section describes the calling convention used by the compiler. These items are examined:

- Function declarations
- C and C++ linkage

- Preserved versus scratch registers
- Function entrance
- Function exit
- Return address handling.

At the end of the section, some examples are shown to describe the calling convention in practice.

Unless otherwise noted, the calling convention used by the compiler adheres to AAPCS, a part of AEABI; see *AEABI compliance*, page 182.

FUNCTION DECLARATIONS

In C, a function must be declared in order for the compiler to know how to call it. A declaration could look as follows:

```
int MyFunction(int first, char * second);
```

This means that the function takes two parameters: an integer and a pointer to a character. The function returns a value, an integer.

In the general case, this is the only knowledge that the compiler has about a function. Therefore, it must be able to deduce the calling convention from this information.

USING C LINKAGE IN C++ SOURCE CODE

In C++, a function can have either C or C++ linkage. To call assembler routines from C++, it is easiest if you make the C++ function have C linkage.

This is an example of a declaration of a function with C linkage:

```
extern "C"
{
    int F(int);
}
```

It is often practical to share header files between C and C++. This is an example of a declaration that declares a function with C linkage in both C and C++:

```
#ifdef __cplusplus
extern "C"
{
#endif

int F(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general ARM CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

Any function is permitted to destroy the contents of a scratch register. If a function needs the register value after a call to another function, it must store it during the call, for example on the stack.

Any of the registers `R0` to `R3`, and `R12`, can be used as a scratch register by the function. Note that `R12` is a scratch register also when calling between assembler functions only because of automatically inserted instructions for veneers.

Preserved registers

Preserved registers, on the other hand, are preserved across function calls. The called function can use the register for other purposes, but must save the value before using the register and restore it at the exit of the function.

The registers `R4` through to `R11` are preserved registers. They are preserved by the called function.

Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer register, `R13/SP`, must at all times point to or below the last element on the stack. In the eventuality of an interrupt, everything below the point the stack pointer points to, can be destroyed. At function entry and exit, the stack pointer must be 8-byte aligned. In the function, the stack pointer must always be word aligned. At exit, `SP` must have the same value as it had at the entry.
- The register `R15/PC` is dedicated for the Program Counter.
- The link register, `R14/LR`, holds the return address at the entrance of the function.

FUNCTION ENTRANCE

Parameters can be passed to a function using one of two basic methods: in registers or on the stack. It is much more efficient to use registers than to take a detour via memory, so the calling convention is designed to use registers as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. These exceptions to the rules apply:

- Interrupt functions cannot take any parameters, except software interrupt functions that accept parameters and have return values

- Software interrupt functions cannot use the stack in the same way as ordinary functions. When an `SVC` instruction is executed, the processor switches to supervisor mode where the supervisor stack is used. Arguments can therefore not be passed on the stack if your application is not running in supervisor mode previous to the interrupt.

Hidden parameters

In addition to the parameters visible in a function declaration and definition, there can be hidden parameters:

- If the function returns a structure larger than 32 bits, the memory location where the structure is to be stored is passed as an extra parameter. Notice that it is always treated as the *first parameter*.
- If the function is a non-static C++ member function, then the `this` pointer is passed as the first parameter (but placed after the return structure pointer, if there is one). For more information, see *Calling assembler routines from C++*, page 138.

Register parameters

The registers available for passing parameters are R0–R3:

Parameters	Passed in registers
Scalar and floating-point values no larger than 32 bits, and single-precision (32-bits) floating-point values	Passed using the first free register: R0–R3
long long and double-precision (64-bit) values	Passed in first available register pair: R0 : R1, or R2 : R3

Table 19: Registers used for passing parameters

The assignment of registers to parameters is a straightforward process. Traversing the parameters from left to right, the first parameter is assigned to the available register or registers. Should there be no more available registers, the parameter is passed on the stack in reverse order.

When functions that have parameters smaller than 32 bits are called, the values are sign or zero extended to ensure that the unused bits have consistent values. Whether the values will be sign or zero extended depends on their type—signed or unsigned.

Stack parameters and layout

Stack parameters are stored in memory, starting at the location pointed to by the stack pointer. Below the stack pointer (towards low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack that is divisible by

four, etc. It is the responsibility of the caller to clean the stack after the called function has returned.

This figure illustrates how parameters are stored on the stack:

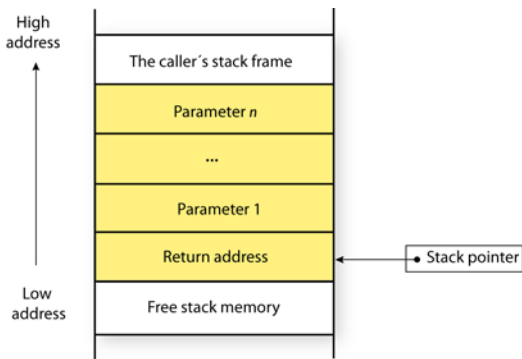


Figure 13: Stack image after the function call

FUNCTION EXIT

A function can return a value to the function or program that called it, or it can have the return type `void`.

The return value of a function, if any, can be scalar (such as integers and pointers), floating-point, or a structure.

Registers used for returning values

The registers available for returning values are `R0` and `R0 : R1`.

Return values	Passed in register/register pair
Scalar and structure return values no larger than 32 bits, and single-precision (32-bit) floating-point return values	R0
The memory address of a structure return value larger than 32 bits	R0
long long and double-precision (64-bit) return values	R0 : R1

Table 20: Registers used for returning values

If the returned value is smaller than 32 bits, the value is sign- or zero-extended to 32 bits.

Stack layout at function exit

It is the responsibility of the caller to clean the stack after the called function has returned.

Return address handling

A function written in assembler language should, when finished, return to the caller, by jumping to the address pointed to by the register `LR`.

At function entry, non-scratch registers and the `LR` register can be pushed with one instruction. At function exit, all these registers can be popped with one instruction. The return address can be popped directly to `PC`.

The following example shows what this can look like:

```
name      call
section   .text:CODE
extern    func

push      {r4-r6,lr}  ; Preserve stack alignment 8
bl        func

; Do something here.

pop       {r4-r6,pc}  ; return
end
```

EXAMPLES

The following section shows a series of declaration examples and the corresponding calling conventions. The complexity of the examples increases toward the end.

Example 1

Assume this function declaration:

```
int add1(int);
```

This function takes one parameter in the register `R0`, and the return value is passed back to its caller in the register `R0`.

This assembler routine is compatible with the declaration; it will return a value that is one number higher than the value of its parameter:

```
name      return
section   .text:CODE
adds     r0, r0, #1
bx       lr
end
```

Example 2

This example shows how structures are passed on the stack. Assume these declarations:

```
struct MyStruct
{
    short a;
    short b;
    short c;
    short d;
    short e;
};

int MyFunction(struct MyStruct x, int y);
```

The values of the structure members `a`, `b`, `c`, and `d` are passed in registers `R0–R3`. The last structure member `e` and the integer parameter `y` are passed on the stack. The calling function must reserve eight bytes on the top of the stack and copy the contents of the two stack parameters to that location. The return value is passed back to its caller in the register `R0`.

Example 3

The function below will return a structure of type `struct MyStruct`.

```
struct MyStruct
{
    int mA;
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in `R0`. The parameter `x` is passed in `R1`.

Assume that the function instead was declared to return a pointer to the structure:

```
struct MyStruct *MyFunction(int x);
```

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in `R0`, and the return value is returned in `R0`.

Call frame information

When you debug an application using C-SPY, you can view the *call stack*, that is, the chain of functions that called the current function. To make this possible, the compiler supplies debug information that describes the layout of the call frame, in particular information about where the return address is stored.

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *ARM® IAR Assembler Reference Guide*.

CFI DIRECTIVES

The `CFI` directives provide C-SPY with information about the state of the calling function(s). Most important of this is the return address, and the value of the stack pointer at the entry of the function or assembler routine. Given this information, C-SPY can reconstruct the state for the calling function, and thereby unwind the stack.

A full description about the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice.

When describing the call frame information, the following three components must be present:

- A *names block* describing the available resources to be tracked
- A *common block* corresponding to the calling convention
- A *data block* describing the changes that are performed on the call frame. This typically includes information about when the stack pointer is changed, and when permanent registers are stored or restored on the stack.

This table lists all the resources defined in the names block used by the compiler:

Resource	Description
CFA R13	The call frames of the stack
R0–R12	Processor general-purpose 32-bit registers
R13	Stack pointer, SP
R14	Link register, LR
S0–S31	Vector Floating Point (VFP) 32-bit coprocessor registers

Table 21: Call frame information resources defined in a names block

Resource	Description
CPSR	Current program status register
SPSR	Saved program status register

Table 21: Call frame information resources defined in a names block (Continued)

CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

The recommended way to create an assembler language routine that handles call frame information correctly is to start with an assembler language source file created by the compiler.

- 1 Start with suitable C source code, for example:

```
int F(int);
int cfiExample(int i)
{
    return i + F(i);
}
```

- 2 Compile the C source code, and make sure to create a list file that contains call frame information—the CFI directives.



On the command line, use the option `-lA`.



In the IDE, choose **Project>Options>C/C++ Compiler>List** and make sure the suboption **Include call frame information** is selected.

For the source code in this example, the list file looks like this:

```
#if Compile
int F(int);
int cfiExample(int i)
{
    return i + F(i);
}
#endif

NAME Cfi

RTMODEL "__SystemLibrary", "DLib"

EXTERN F

PUBLIC cfiExample

CFI Names cfiNames0
CFI StackFrame CFA R13 DATA
CFI Resource R0:32, R1:32, R2:32, R3:32, R4:32, \
```

```

R5:32, R6:32, R7:32
CFI Resource R8:32, R9:32, R10:32, R11:32, R12:32, \
R13:32, R14:32
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 4
CFI DataAlign 4
CFI ReturnAddress R14 CODE
CFI CFA R13+0
CFI R0 Undefined
CFI R1 Undefined
CFI R2 Undefined
CFI R3 Undefined
CFI R4 SameValue
CFI R5 SameValue
CFI R6 SameValue
CFI R7 SameValue
CFI R8 SameValue
CFI R9 SameValue
CFI R10 SameValue
CFI R11 SameValue
CFI R12 Undefined
CFI R14 SameValue
CFI EndCommon cfiCommon0

SECTION `.text`:CODE:NOROOT(2)
CFI Block cfiBlock0 Using cfiCommon0
CFI Function cfiExample
ARM
cfiExample:
    PUSH    {R4,LR}
    CFI R14 Frame(CFA, -4)
    CFI R4 Frame(CFA, -8)
    CFI CFA R13+8
    MOVS    R4,R0
    MOVS    R0,R4
    BL      F
    ADDS    R0,R0,R4
    POP     {R4,PC}      ;; return
    CFI EndBlock cfiBlock0

END

```

Note: The header file `Common.i` contains the macros `CFI_NAME_BLOCK`, `CFI_COMMON_ARM`, and `CFI_COMMON_Thumb`, which declare a typical names block and

two typical common blocks. These two macros declare several resources, both concrete and virtual.

Using C

This chapter gives an overview of the compiler's support for the C language. The chapter also gives a brief overview of the IAR C language extensions.

C language overview

The IAR C/C++ Compiler for ARM supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

In addition, the compiler also supports the ISO 9899:1990 standard (including all technical corrigenda and addenda), also known as C94, C90, C89, and ANSI C. In this guide, this standard is referred to as *C89*. Use the `--c89` compiler option to enable this standard.

The C99 standard is derived from C89, but adds features like these:

- The `inline` keyword advises the compiler that the function declared immediately after the directive should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a `for` loop
- The `bool` data type
- The `long long` data type
- The `complex` floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator `_Pragma()`
- Variadic macros, which are the preprocessor macro equivalents of `printf` style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option `--vla`
- Inline assembler using the `asm` or the `__asm` keyword, see *Inline assembler*, page 131.

Note: Even though it is a C99 feature, the IAR C/C++ Compiler for ARM does not support UCNs (universal character names).

Extensions overview

The compiler offers the features of Standard C and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

This is an overview of the available extensions:

- IAR C language extensions

For information about available language extensions, see *IAR C language extensions*, page 153. For more information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

The `#pragma` directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages.

Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For information about available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

The preprocessor of the compiler adheres to Standard C. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. For more information about using intrinsic functions, see *Mixing C and assembler*, page 129. For information about available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides the C and C++ library definitions that apply to embedded systems. For more information, see *IAR DLIB Library*, page 375.

Note: Any use of these extensions, except for the pragma directives, makes your source code inconsistent with Standard C.

ENABLING LANGUAGE EXTENSIONS

You can choose different levels of language conformance by means of project options:

Command line	IDE*	Description
--strict	Strict	All <i>IAR C language extensions</i> are disabled; errors are issued for anything that is not part of Standard C.
None	Standard	All <i>extensions to Standard C</i> are enabled, but no <i>extensions for embedded systems programming</i> . For information about extensions, see <i>IAR C language extensions</i> , page 153.
-e	Standard with IAR extensions	All <i>IAR C language extensions</i> are enabled.

Table 22: Language extensions

* In the IDE, choose **Project>Options> C/C++ Compiler>Language>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific core you are using, typically to meet memory restrictions
- *Relaxations to Standard C*—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions, see *Relaxations to Standard C*, page 156.

EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

The following language extensions are available both in the C and the C++ programming languages and they are well suited for embedded systems programming:

- Type attributes, and object attributes
For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.
- Placement at an absolute address or in a named section
The @ operator or the directive #pragma location can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named section. For more information about using these features, see *Controlling data and function placement in memory*, page 191, and *location*, page 326.
- Alignment control
Each data type has its own alignment; for more information, see *Alignment*, page 287. If you want to change the alignment, the __packed data type attribute, and the #pragma pack, and #pragma data_alignment directives are available. If you want to check the alignment of an object, use the __ALIGNOF__() operator.
The __ALIGNOF__ operator is used for accessing the alignment of an object. It takes one of two forms:
 - __ALIGNOF__ (type)
 - __ALIGNOF__ (expression)
 In the second form, the expression is not evaluated.
- Anonymous structs and unions
C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 189.
- Bitfields and non-standard types
In Standard C, a bitfield must be of the type int or unsigned int. Using IAR C language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. For more information, see *Bitfields*, page 290.
- static_assert()
The construction static_assert(const-expression, "message"); can be used in C/C++. The construction will be evaluated at compile time and if const-expression is false, a message will be issued including the message string.

- Parameters in variadic macros
Variadic macros are the preprocessor macro equivalents of `printf` style functions. The preprocessor accepts variadic macros with no arguments, which means if no parameter matches the `...` parameter, the comma is then deleted in the `" , ##__VA_ARGS__"` macro definition. According to Standard C, the `...` parameter must be matched with at least one argument.

Dedicated section operators

The compiler supports getting the start address, end address, and size for a section with these built-in section operators:

<code>__section_begin</code>	Returns the address of the first byte of the named section or block.
<code>__section_end</code>	Returns the address of the first byte <i>after</i> the named section or block.
<code>__section_size</code>	Returns the size of the named section or block in bytes.

Note: The aliases `__segment_begin/__sfb`, `__segment_end/__sfe`, and `__segment_size/__sfs` can also be used.

The operators can be used on named sections or on named blocks defined in the linker configuration file.

These operators behave syntactically as if declared like:

```
void * __section_begin(char const * section)
void * __section_end(char const * section)
size_t * __section_size(char const * section)
```

When you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined section, or when you use named blocks in the linker configuration file, the section operators can be used for getting the start and end address of the memory range where the sections or blocks were placed.

The named *section* must be a string literal and it must have been declared earlier with the `#pragma section` directive. The type of the `__section_begin` operator is a pointer to `void`. Note that you must enable language extensions to use these operators.

The operators are implemented in terms of *symbols* with dedicated names, and will appear in the linker map file under these names:

Operator	Symbol
<code>__section_begin(sec)</code>	<code>sec\$\$Base</code>

Table 23: Section operators and their symbols

Operator	Symbol
<code>__section_end(sec)</code>	<code>sec\$\$Limit</code>
<code>__section_size(sec)</code>	<code>sec\$\$Length</code>

Table 23: Section operators and their symbols (Continued)

Note that the linker will not necessarily place sections with the same name consecutively when these operators are not used. Using one of these operators (or the equivalent symbols) will cause the linker to behave as if the sections were in a named block. This is to assure that the sections are placed consecutively, so that the operators can be assigned meaningful values. If this is in conflict with the section placement as specified in the linker configuration file, the linker will issue an error.

Example

In this example, the type of the `__section_begin` operator is `void *`.

```
#pragma section="MYSECTION"
...
section_start_address = __section_begin("MYSECTION");
```

See also *section*, page 331, and *location*, page 326.

RELAXATIONS TO STANDARD C

This section lists and briefly describes the relaxation of some Standard C issues and also some useful but minor syntax extensions:

- Arrays of incomplete types
An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).
- Forward declaration of `enum` types
The extensions allow you to first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.
- Accepting missing semicolon at the end of a `struct` or `union` specifier
A warning—instead of an error—is issued if the semicolon at the end of a `struct` or `union` specifier is missing.
- Null and `void`
In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In Standard C, some operators allow this kind of behavior, while others do not allow it.

- Casting pointers to integers in static initializers
In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 296.
- Taking the address of a register variable
In Standard C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.
- `long float` means `double`
The type `long float` is accepted as a synonym for `double`.
- Repeated `typedef` declarations
Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.
- Mixing pointer types
Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.
Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.
- Non-top level `const`
Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.
- Non-lvalue arrays
A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.
- Comments at the end of preprocessor directives
This extension, which makes it legal to place text after preprocessor directives, is enabled unless the strict Standard C mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.
- An extra comma at the end of `enum` lists
Placing an extra comma is allowed at the end of an `enum` list. In strict Standard C mode, a warning is issued.
- A label preceding a `}`
In Standard C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler allows this, but issues a warning.

Note that this also applies to the labels of `switch` statements.

- Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

Standard C requires that all initializer expressions of static arrays, structs, and unions are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:

```
struct str
{
    int a;
} x = 10;
```

- Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

- Expanding function names into strings with the function as context

Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string that contains the name of the current function. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char)"
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see *-e*, page 235.

- Static functions in function and block scopes

Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- Numbers scanned according to the syntax for numbers

Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one valid token. (If the `--strict` option is used, the `pp-number` syntax is used instead.)

Using C++

IAR Systems supports the C++ language. You can choose between these standards: Standard C++, the industry-standard Embedded C++, and Extended Embedded C++. This chapter describes what you need to consider when using the C++ language.

Overview—Embedded C++ and Extended Embedded C++

Embedded C++ is a proper subset of the ISO/IEC C++ standard which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language. EC++ offers the same object-oriented benefits as C++, but without some features that can increase code size and execution time in ways that are hard to predict.

EMBEDDED C++

These C++ features are supported:

- Classes, which are user-defined types that incorporate both data structure and behavior; the essential feature of inheritance allows data structure and behavior to be shared among classes
- Polymorphism, which means that an operation can behave differently on different classes, is provided by virtual functions
- Overloading of operators and function names, which allows several operators or functions with the same name, provided that their argument lists are sufficiently different
- Type-safe memory management using the operators `new` and `delete`
- Inline functions, which are indicated as particularly suitable for inline expansion.

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are features added very late before Standard C++ was defined. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

Embedded C++ lacks these features of C++:

- Templates
- Multiple and virtual inheritance
- Exception handling

- Runtime type information
- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

Note: The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

EXTENDED EMBEDDED C++

IAR Systems' Extended EC++ is a slightly larger subset of C++ which adds these features to the standard EC++:

- Full template support
- Multiple and virtual inheritance
- Namespace support
- The `mutable` attribute
- The cast operators `static_cast`, `const_cast`, and `reinterpret_cast`.

All these added features conform to the C++ standard.

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++ language, which means no exceptions, no multiple inheritance, and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

Note: A module compiled with Extended EC++ enabled is fully link-compatible with a module compiled without Extended EC++ enabled.

Overview—Standard C++

The IAR C++ implementation fully complies with the ISO/IEC 1488:2003 C++ standard. In this guide, this standard is referred to as Standard C++.

The main reason for using Standard C++ instead of EC++ or EEC++ is when there is a need for either:

- Exception support
- Runtime type information (RTTI) support
- The standard C++ library (the EC++ library is a stripped version of the C++ library where streams and strings are not templates).

If code size is important and your application does not need any of these features, EC++ (or EEC++) is the better choice.

MODES FOR EXCEPTIONS AND RTTI SUPPORT

Both exceptions and runtime type information result in increased code size simply by being included in your application. You might want to disable either or both of these features to avoid this increase:

- Support for runtime type information constructs can be disabled by using the compiler option `--no_rtti`
- Support for exceptions can be disabled by using the compiler option `--no_exceptions`

Even if support is enabled while compiling, the linker can avoid including the extra code and tables in the final application. If no part of your application actually throws an exception, the code and tables supporting the use of exceptions are not included in the application code image. Also, if dynamic runtime type information constructs (`dynamic_cast/typeid`) are not used with polymorphic types, the objects needed to support them are not included in the application code image. To control this behavior, use the linker options `--no_exceptions`, `--force_exceptions`, and `--no_dynamic_rtti_elimination`.

Disabling exception support

When you use the compiler option `--no_exceptions`, the following will generate a compiler error:

- `throw` expressions
- `try-catch` statements
- Exception specifications on function definitions.

In addition, the extra code and tables needed to handle destruction of objects with auto storage duration when an exception is propagated through a function will not be generated when the compiler option `--no_exceptions` is used.

All functionality in system headers not directly involving exceptions is supported when the compiler option `--no_exceptions` is used.

The linker will produce an error if you try to link C++ modules compiled with exception support with modules compiled without exception support

For more information, see *--no_exceptions*, page 244.

Disabling RTTI support

When you use the compiler option `--no_rtti`, the following will generate a compiler error:

- The `typeid` operator
- The `dynamic_cast` operator.

Note: If `--no_rtti` is used but exception support is enabled, most RTTI support is still included in the compiler output object file because it is needed for exceptions to work.

For more information, see *--no_rtti*, page 246.

EXCEPTION HANDLING

Exception handling can be divided into three parts:

- Exception raise mechanisms—in C++ they are the `throw` and `rethrow` expressions.
- Exception catch mechanisms—in C++ they are the `try-catch` statements, the exception specifications for a function, and the implicit catch to prevent an exception leaking out from `main`.
- Information about currently active functions—if they have `try-catch` statements and the set of auto objects whose destructors need to be run if an exception is propagated through the function.

When an exception is raised, the function call stack is unwound, function by function, block by block. For each function or block, the destructors of auto objects that need destruction are run, and a check is made whether there is a catch handler for the exception. If there is, the execution will continue from that catch handler.

An application that mixes C++ code with assembler and C code, and that throws exceptions from one C++ function to another via assembler routines and C functions must use the linker option `--exception_tables` with the argument `unwind`.

The implementation of exceptions

Exceptions are implemented using a table method. For each function, the tables describe:

- How to unwind the function, that is, how to find its caller on the stack and restore registers that need restoring
- Which catch handlers that exist in the function
- Whether the function has an exception specification and which exceptions it allows to propagate
- The set of auto objects whose destructors must be run.

When an exception is raised, the runtime will proceed in two phases. The first phase will use the exception tables to search the stack for a function invocation containing a catch handler or exception specification that would cause stack unwinding to halt at that point. Once this point is found, the second phase is entered, doing the actual unwinding, and running the destructors of auto objects where that is needed.

The table method results in virtually no overhead in execution time or RAM usage when an exception is not actually thrown. It does incur a significant penalty in read-only memory usage for the tables and the extra code, and throwing and catching an exception is a relatively expensive operation.

The destruction of auto objects when the stack is being unwound as a result of an exception is implemented in code separated from the code that handles the normal operation of a function. This code, together with the code in catch handlers, is placed in a separate section (`.exc.text`) from the normal code (normally placed in `.text`). In some cases, for instance when there is fast and slow ROM memory, it can be advantageous to select on this difference when placing sections in the linker configuration file.

Enabling support for C++ and variants



In the compiler, the default language is C.

To compile files written in Standard C++, you must use the `--c++` compiler option. See `--c++`, page 229.

To compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 235.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 235.



To enable EC++, EEC++, or C++ in the IDE, choose **Project>Options>C/C++ Compiler>Language** and select the appropriate standard.

C++ and EC++ feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for ARM, you must be aware of some benefits and some possible quirks when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

USING IAR ATTRIBUTES WITH CLASSES

Static data members of C++ classes are treated the same way global variables are, and can have any applicable IAR type and object attribute.

Member functions are in general treated the same way free functions are, and can have any applicable IAR type and object attributes. Virtual member functions can only have attributes that are compatible with default function pointers, and constructors and destructors cannot have any such attributes.

The location operator @ and the #pragma location directive can be used on static data members and with all member functions.

Example

```
class MyClass
{
public:
    // Locate a static variable in __memattr memory at address 60
    static __no_init int mI @ 60;

    // A static Thumb function
    static __thumb void F();

    // A Thumb function
    __thumb void G();

    // Interworking assumed
    virtual __arm void ArmH();

    // Interworking assumed
    virtual __thumb void ThumbH();

    // Locate a virtual function into SPECIAL
    virtual void M() const volatile @ "SPECIAL";
};
```

FUNCTION TYPES

A function type with `extern "C"` linkage is compatible with a function that has C++ linkage.

Example

```
extern "C"
{
    typedef void (*FpC)(void);    // A C function typedef
}

typedef void (*FpCpp)(void);    // A C++ function typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
    MyF(F1);                      // Always works
    MyF(F2);                      // FpCpp is compatible with FpC
}
```

USING STATIC CLASS OBJECTS IN INTERRUPTS

If interrupt functions use static class objects that need to be constructed (using constructors) or destroyed (using destructors), your application will not work properly if the interrupt occurs before the objects are constructed, or, during or after the objects are destroyed.

To avoid this, make sure that these interrupts are not enabled until the static objects have been constructed, and are disabled when returning from `main` or calling `exit`. For information about system startup, see *System startup and termination*, page 107.

Function local static class objects are constructed the first time execution passes through their declaration, and are destroyed when returning from `main` or when calling `exit`.

USING NEW HANDLERS

To handle memory exhaustion, you can use the `set_new_handler` function.

New handlers in Standard C++

If you do not call `set_new_handler`, or call it with a `NULL` new handler, and `operator new` fails to allocate enough memory, `operator new` will throw `std::bad_alloc` if exceptions are enabled. If exceptions are not enabled, `operator new` will instead call `abort`.

If you call `set_new_handler` with a non-NULL new handler, the provided new handler will be called by `operator new` if the `operator new` fails to allocate enough memory. The new handler must then make more memory available and return, or abort execution in some manner. If exceptions are enabled, the new handler can also throw a `std::bad_alloc` exception. The `nothrow` variant of `operator new` will only return NULL in the presence of a new handler if exceptions are enabled and the new handler throws `std::bad_alloc`.

New handlers in Embedded C++

If you do not call `set_new_handler`, or call it with a NULL new handler, and `operator new` fails to allocate enough memory, it will call `abort`. The `nothrow` variant of the new operator will instead return NULL.

If you call `set_new_handler` with a non-NULL new handler, the provided new handler will be called by `operator new` if `operator new` fails to allocate memory. The new handler must then make more memory available and return, or abort execution in some manner. The `nothrow` variant of `operator new` will never return NULL in the presence of a new handler.

TEMPLATES

C++ and Extended EC++ supports templates according to the C++ standard, but not the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

DEBUG SUPPORT IN C-SPY

C-SPY® has debug support for some features in C++.

- C-SPY® has built-in display support for the STL containers. The logical structure of containers is presented in the watch views in a comprehensive way that is easy to understand and follow.
- You can make C-SPY stop at a `throw` statement or if a raised exception does not have any corresponding `catch` statement.

For more information about this, see the *C-SPY® Debugging Guide for ARM®*.

EEC++ feature description

This section describes features that distinguish Extended EC++ from EC++.

TEMPLATES

The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, see *Extended Embedded C++*, page 162.

VARIANTS OF CAST OPERATORS

In Extended EC++ these additional variants of C++ cast operators can be used:

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

NAMESPACE

The namespace feature is only supported in *Extended* EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

EC++ and C++ language extensions

When you use the compiler in any C++ mode and enable IAR language extensions, the following C++ language extensions are available in the compiler

- In a friend declaration of a class, the `class` keyword can be omitted, for example:

```
class B;
class A
{
    friend B;           //Possible when using IAR language
                        //extensions
    friend class B; //According to the standard
};
```

- Constants of a scalar type can be defined within classes, for example:

```
class A
{
    const int mSize = 10; //Possible when using IAR language
                        //extensions
    int mArr[mSize];
};
```

According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:

```
struct A
{
    int A::F(); // Possible when using IAR language extensions
    int G();    // According to the standard
};
```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

```
extern "C" void F(); // Function with C linkage
void (*PF)()         // PF points to a function with C++ linkage
    = &F; // Implicit conversion of function pointer.
```

According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```
bool X;

char *P1 = X ? "abc" : "def";           //Possible when using IAR
                                        //language extensions
char const *P2 = X ? "abc" : "def"; //According to the standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.
- An anonymous union can be introduced into a containing class by a `typedef` name. It is not necessary to first declare the union. For example:

```
typedef union
{
    int i,j;
} U; // U identifies a reusable anonymous union.

class A
{
public:
    U; // OK -- references to A::i and A::j are allowed.
};
```

In addition, this extension also permits *anonymous classes* and *anonymous structs*, as long as they have no C++ features (for example, no static data members or member functions, and no non-public members) and have no nested types other than other anonymous classes, structs, or unions. For example:

```
struct A
{
    struct
    {
        int i,j;
    }; // OK -- references to A::i and A::j are allowed.
};
```

- The friend class syntax allows nonclass types as well as class types expressed through a `typedef` without an elaborated type name. For example:

```
typedef struct S ST;

class C
{
public:
    friend S; // Okay (requires S to be in scope)
    friend ST; // Okay (same as "friend S;")
    // friend S const; // Error, cv-qualifiers cannot
                        // appear directly
};
```

Note: If you use any of these constructions without first enabling language extensions, errors are issued.

Application-related considerations

This chapter discusses a selected range of application issues related to developing your embedded application.

Typically, this chapter highlights issues that are not specifically related to only the compiler or the linker.

Output format considerations

The linker produces an absolute executable image in the ELF/DWARF object file format.

You can use the IAR ELF Tool—`ielftool`—to convert an absolute ELF image to a format more suitable for loading directly to memory, or burning to a PROM or flash memory etc.

`ielftool` can produce these output formats:

- Plain binary
- Motorola S-records
- Intel hex.

Note: `ielftool` can also be used for other types of transformations, such as filling and calculating checksums in the absolute image.

The source code for `ielftool` is provided in the `arm/src` directory. For more information about `ielftool`, see *The IAR ELF Tool—ielftool*, page 420.

Stack considerations

The stack is used by functions to store variables and other information that is used locally by functions, as described in the chapter *Data storage*. It is a continuous block of memory pointed to by the processor stack pointer register `SP`.

The block used for holding the stack is called `CSTACK`. The system startup code initializes the stack pointer to the end of the stack.

STACK SIZE CONSIDERATIONS

The compiler uses the internal data stack for a variety of user application operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too large, RAM will be wasted. If the given stack size is too small, two things can happen, depending on where in memory you located your stack. Both alternatives are likely to result in application failure. Either variable storage will be overwritten, leading to undefined behavior, or the stack will fall outside of the memory area, leading to an abnormal termination of your application. Because the second alternative is easier to detect, you should consider placing your stack so that it grows toward the end of the memory.

For more information about the stack size, see *Setting up the stack, page 85*, and *Saving stack space and RAM memory, page 201*.

STACK ALIGNMENT

The default `cstartup` code automatically initializes all stacks to an 8-byte aligned address.

For more information about aligning the stack, see *Calling convention, page 139* and more specifically *Special registers, page 141* and *Stack parameters and layout, page 142*.

EXCEPTION STACKS

The ARM architecture supports five exception modes which are entered when different exceptions occur. Each exception mode has its own stack to avoid corrupting the System/User mode stack.

The table shows proposed stack names for the various exception stacks, but any name can be used:

Processor mode	Proposed stack section name	Description
Supervisor	SVC_STACK	Operating system stack.
IRQ	IRQ_STACK	Stack for general-purpose (IRQ) interrupt handlers.
FIQ	FIQ_STACK	Stack for high-speed (FIQ) interrupt handlers.
Undefined	UND_STACK	Stack for undefined instruction interrupts. Supports software emulation of hardware coprocessors and instruction set extensions.

Table 24: Exception stacks

Processor mode	Proposed stack section name	Description
Abort	ABT_STACK	Stack for instruction fetch and data access memory abort interrupt handlers.

Table 24: Exception stacks (Continued)

For each processor mode where a stack is needed, a separate stack pointer must be initialized in your startup code, and section placement should be done in the linker configuration file. The IRQ and FIQ stacks are the only exception stacks which are preconfigured in the supplied `cstartup.s` and `lnkarm.icf` files, but other exception stacks can easily be added.

Cortex-M does not have individual exception stacks. By default, all exception stacks are placed in the `CSTACK` section.



To view any of these stacks in the Stack window available in the IDE, these preconfigured section names must be used instead of user-defined section names.

Heap considerations

The heap contains dynamic data allocated by use of the C function `malloc` (or one of its relatives) or the C++ operator `new`.

If your application uses dynamic memory allocation, you should be familiar with:

- Linker sections used for the heap
- Allocating the heap size, see *Setting up the heap*, page 86.

The memory allocated to the heap is placed in the section `HEAP`, which is only included in the application if dynamic memory allocation is actually used.



Heap size and standard I/O

If you excluded `FILE` descriptors from the `DLIB` runtime environment, as in the normal configuration, there are no input and output buffers at all. Otherwise, as in the full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an ARM core. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

Interaction between the tools and your application

The linking process and the application can interact symbolically in four ways:

- Creating a symbol by using the ILINK command line option `--define_symbol`. ILINK will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.
- Creating an exported configuration symbol by using the command line option `--config_def` or the configuration directive `define symbol`, and exporting the symbol using the `export symbol` directive. ILINK will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.

One advantage of this symbol definition is that this symbol can also be used in expressions in the configuration file, for example to control the placement of sections into memory ranges.

- Using the compiler operators `__section_begin`, `__section_end`, or `__section_size`, or the assembler operators `SFB`, `SFE`, or `SIZEOF` on a named section or block. These operators provide access to the start address, end address, and size of a contiguous sequence of sections with the same name, or of a linker block specified in the linker configuration file.
- The command line option `--entry` informs ILINK about the start label of the application. It is used by ILINK as a root symbol and to inform the debugger where to start execution.

The following lines illustrate how to use these mechanisms. Add these options to your command line:

```
--define_symbol NumberOfElements=10
--config_def HeapSize=1024
```

The linker configuration file can look like this:

```
define memory Mem with size = 4G;
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* Export of symbol */
export symbol HeapSize;

/* Setup a heap area with a size defined by an ILINK option */
define block MyHEAP with size = HeapSize, alignment = 8 {};

place in RAM { block MyHEAP };
```


Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by ILINK option to dynamically allocate an
array of elements with specified size. The value takes the form
of a label.
*/
extern int NrOfElements;

typedef char Elements;
Elements *GetElementArray()
{
    return malloc(sizeof(Elements) * (long) &NrOfElements);
}

/* Use a symbol defined by ILINK option, a symbol that in the
* configuration file was made available to the application.
*/
extern char HeapSize;

/* Declare the section that contains the heap. */
#pragma section = "MYHEAP"

char *MyHeap()
{
    /* First get start of statically allocated section, */
    char *p = __section_begin("MYHEAP");

    /* ...then we zero it, using the imported size. */
    for (int i = 0; i < (int) &HeapSize; ++i)
    {
        p[i] = 0;
    }
    return p;
}
```

Checksum calculation

The IAR ELF Tool—`ielftool`—fills specific ranges of memory with a pattern and then calculates a checksum for those ranges. The calculated checksum replaces the value of an existing symbol in the input ELF image. The application can then verify that the ranges did not change.

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

- Reserve a place, with an associated name and size, for the checksum calculated by `ielftool`
- Choose a checksum algorithm, set up `ielftool` for it, and include source code for the algorithm in your application
- Decide what memory ranges to verify and set up both `ielftool` and the source code for it in your application source code.



Note: To set up `ielftool` in the IDE, choose **Project>Options>Linker>Checksum**.

CALCULATING A CHECKSUM

In this example, a checksum is calculated for ROM memory at 0x8002 up to 0x8FFF and the 2-byte calculated checksum is placed at 0x8000.

Creating a place for the calculated checksum

You can create a place for the calculated checksum in two ways; by creating a global C/C++ or assembler constant symbol with a proper size, residing in a specific section (in this example `.checksum`), or by using the linker option `--place_holder`.

For example, to create a 2-byte space for the symbol `__checksum` in the section `.checksum`, with alignment 4:

```
--place_holder __checksum,2,.checksum,4
```

Note: The `.checksum` section will only be included in your application if the section appears to be needed. If the checksum is not needed by the application itself, you can use the linker option `--keep=__checksum` or the linker directive `keep` to force the section to be included.

To place the `.checksum` section, you must modify the linker configuration file. It can look like this (note the handling of the block `CHECKSUM`):

```
define memory Mem with size = 4G;

define region ROM_region = Mem:[from 0x8000 to 0x80000000 - 1];
define region RAM_region = Mem:[from 0x80000000 to 0x100000000 - 2
];

initialize by copy { rw };
do not initialize { section .noinit };

define block HEAP      with alignment = 8, size = 16M {};
define block CSTACK    with alignment = 8, size = 16K {};
define block IRQ_STACK with alignment = 8, size = 16K {};
define block FIQ_STACK with alignment = 8, size = 16K {};
```

```

define block CHECKSUM      { ro section .checksum };
place at address Mem:0x0 { ro section .intvec};
place in ROM_region { ro, first block CHECKSUM };
place in RAM_region { rw, block HEAP, block CSTACK, block
                        IRQ_STACK, block FIQ_STACK };

```

Running ielftool

To calculate the checksum, run `ielftool`:

```

ielftool --fill=0x00;0x8000-0x8FFF
--checksum=__checksum:2,crc16;0x8000-0x8FFF sourceFile.out
destinationFile.out

```

To calculate a checksum you also must define a fill operation. In this example, the fill pattern `0x0` is used. The checksum algorithm used is `crc16`.

Note that `ielftool` needs an unstripped input ELF image. If you use the `--strip` linker option, remove it and use the `--strip ielftool` option instead.

ADDING A CHECKSUM FUNCTION TO YOUR SOURCE CODE

To check the value of the `ielftool` generated checksum, it must be compared with a checksum that your application calculated. This means that you must add a function for checksum calculation (that uses the same algorithm as `ielftool`) to your application source code. 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 SlowCrc16(unsigned 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 ^= 0x1021;
        byte <= 1;
    }
}
return sum;
}

```

You can find the source code for the checksum algorithms in the `arm\src\linker` directory of your product installation.

Checksum calculation

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

```

/* The checksum calculated by ielftool
 * (note that it is located on address 0x8000)
 */
extern unsigned short const __checksum;

void TestChecksum()
{
    unsigned short calc = 0;
    unsigned char  zeros[2] = {0, 0};

    /* Run the checksum algorithm */
    calc = SlowCrc16(0,
                    (unsigned char *) checksumStart,
                    (checksumEnd - checksumStart+1));

    /* Rotate out the answer */
    calc = SlowCrc16(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 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 a slow function variant is used, you must make a final call to the checksum calculation with as many bytes (with the value 0x00) as there are bytes in the checksum.

For more information, see also *The IAR ELF Tool—ielftool*, page 420.

C-SPY CONSIDERATIONS

By default, a symbol that you have allocated in memory by using the linker option `--place_holder` is considered by C-SPY to be of the type `int`. If the size of the checksum is different than the size of an `int`, you can change the display format of the checksum symbol to match its size.



In the C-SPY Watch window, select the symbol and choose **Show As** from the context menu. Choose the display format that matches the size of the checksum symbol.

Linker optimizations

VIRTUAL FUNCTION ELIMINATION

Virtual Function Elimination (VFE) is a linker optimization that removes unneeded virtual functions and dynamic runtime type information.

In order for Virtual Function Elimination to work, all relevant modules must provide information about virtual function table layout, which virtual functions are called, and for which classes dynamic runtime type information is needed. If one or more modules do not provide this information, a warning is generated by the linker and Virtual Function Elimination is not performed.

If you know that modules that lack such information do not perform any virtual function calls and do not define any virtual function tables, you can use the `--vfe=forced` linker option to enable Virtual Function Elimination anyway.

Currently, tools from IAR Systems and from RealView provide the information needed for Virtual Function Elimination in a way that the linker can use.

Note that you can disable Virtual Function Elimination entirely by using the `--no_vfe` linker option. In this case, no warning will be issued for modules that lack VFE information.

For more information, see `--vfe`, page 285 and `--no_vfe`, page 280.

AEABI compliance

The IAR build tools for ARM support the Embedded Application Binary Interface for ARM, AEABI, defined by ARM Limited. This interface is based on the Intel IA64 ABI interface. The advantage of adhering to AEABI is that any such module can be linked with any other AEABI-compliant module, even modules produced by tools provided by other vendors.

The IAR build tools for ARM support the following parts of the AEABI:

AAPCS	Procedure Call Standard for the ARM architecture
CPPABI	C++ ABI for the ARM architecture (EC++ parts only)
AAELF	ELF for the ARM architecture
AADWARF	DWARF for the ARM architecture
RTABI	Runtime ABI for the ARM architecture
CLIBABI	C library ABI for the ARM architecture

The IAR build tools only support a *bare metal* platform, that is a ROM-based system that lacks an explicit operating system.

Note that:

- The AEABI is specified for C89 only
- The IAR build tools only support using the default and C locales
- The AEABI does not specify C++ library compatibility
- Neither the size of an `enum` or of `wchar_t` is constant in the AEABI.

If AEABI compliance is enabled, almost all optimizations performed in the system header files are turned off, and certain preprocessor constants become real constant variables instead.

LINKING AEABI-COMPLIANT MODULES USING THE IAR ILINK LINKER

When building an application using the IAR ILINK Linker, the following types of modules can be combined:

- Modules produced using IAR build tools, both AEABI-compliant modules as well as modules that are not AEABI-compliant
- AEABI-compliant modules produced using build tools from another vendor.

Note: To link a module produced by a compiler from another vendor, extra support libraries from that vendor might be required.

The IAR ILINK Linker automatically chooses the appropriate standard C/C++ libraries to use based on attributes from the object files. Imported object files might not have all these attributes. Therefore, you might need to help ILINK choose the standard library by verifying one or more of the following details:

- The used CPU by specifying the `--cpu` linker option
- If full I/O is needed; make sure to link with a Full library configuration in the standard library
- Explicitly specify runtime library file(s), possibly in combination with the `--no_library_search` linker option.

When linking, you should also consider virtual function elimination, see *Virtual Function Elimination*, page 181.

LINKING AEABI-COMPLIANT MODULES USING A THIRD-PARTY LINKER

If you have a module produced using the IAR C/C++ Compiler and you plan to link that module using a linker from a different vendor, that module must be AEABI-compliant, see *Enabling AEABI compliance in the compiler*, page 183.

In addition, if that module uses any of the IAR-specific compiler extensions, you must make sure that those features are also supported by the tools from the other vendor. Note specifically:

- Support for the following extensions must be verified: `#pragma pack`, `__no_init`, `__root`, and `__ramfunc`
- The following extensions are harmless to use: `#pragma location/@`, `__arm`, `__thumb`, `__swi`, `__irq`, `__fiq`, and `__nested`.

ENABLING AEABI COMPLIANCE IN THE COMPILER

You can enable AEABI compliance in the compiler by setting the `--aeabi` option. In this case, you must also use the `--guard_calls` option.



In the IDE, use the **Project>Options>C/C++ Compiler>Extra Options** page to specify the `--aeabi` and `--guard_calls` options.



On the command line, use the options `--aeabi` and `--guard_calls` to enable AEABI support in the compiler.

Alternatively, to enable support for AEABI for a specific system header file, you must define the preprocessor symbol `_AEABI_PORTABILITY_LEVEL` to non-zero prior to including a system header file, and make sure that the symbol `AEABI_PORTABLE` is set to non-zero after the inclusion of the header file:

```
#define _AEABI_PORTABILITY_LEVEL 1
#undef _AEABI_PORTABLE
#include <header.h>
#ifdef _AEABI_PORTABLE
    #error "header.h not AEABI compatible"
#endif
```

CMSIS integration

The `arm\CMSIS` subdirectory contains CMSIS (ARM Cortex Microcontroller Software Interface Standard) and CMSIS DSP header and library files, and documentation. For more information about CMSIS, see <http://www.arm.com/cmsis>.

The special header file `inc\c\cmsis_iar.h` is provided as a CMSIS adaptation of the current version of the IAR C/C++ Compiler.

CMSIS DSP LIBRARY

IAR Embedded Workbench comes with prebuilt CMSIS DSP libraries in the `arm\CMSIS\Lib\IAR` directory. The names of the library files are constructed in this way:

```
iar_cortexM<0|3|4><1|b>[f]_math.a
```

where `<0|3|4>` selects the Cortex-M variant, `<1|b>` selects the byte order, and `[f]` indicates that the library is built for FPU (Cortex-M4 only).

CUSTOMIZING THE CMSIS DSP LIBRARY

The source code of the CMSIS DSP library is provided in the `arm\CMSIS\DSP_Lib\Source` directory. You can find an IAR Embedded Workbench project which is prepared for building a customized DSP library in the `arm\CMSIS\DSP_Lib\Source\IAR` directory.



BUILDING WITH CMSIS ON THE COMMAND LINE

This section contains examples of how to build your CMSIS-compatible application on the command line.

CMSIS only (that is without the DSP library)

```
icccarm -I $EW_DIR$\arm\CMSIS\Include
```

With the DSP library, for Cortex-M4, little-endian, and with FPU

```
icccarm --endian=little --cpu=Cortex-M4 --fpu=VFPv4_sp -I  
$EW_DIR$\arm\CMSIS\Include -D ARM_MATH_CM4
```

```
ilinkarm $EW_DIR$\arm\CMSIS\Lib\IAR\iar_cortexM4lf_math.a
```

With the DSP library, for Cortex-M3, and little-endian

```
icccarm --endian=little --cpu=Cortex-M3 -I  
$EW_DIR$\arm\CMSIS\Include -D ARM_MATH_CM3
```

```
ilinkarm $EW_DIR$\arm\CMSIS\Lib\IAR\iar_cortexM3l_math.a
```



BUILDING WITH CMSIS IN IAR EMBEDDED WORKBENCH

Choose **Project>Options>General Options>Library Configuration** to enable CMSIS support.

When enabled, CMSIS include paths and the DSP library will automatically be used. For more information, see the *IDE Project Management and Building Guide for ARM®*.

Efficient coding for embedded applications

For embedded systems, the size of the generated code and data is very important, because using smaller external memory or on-chip memory can significantly decrease the cost and power consumption of a system.

The topics discussed are:

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Facilitating good code generation.

As a part of this, the chapter also demonstrates some of the more common mistakes and how to avoid them, and gives a catalog of good coding techniques.

Selecting data types

For efficient treatment of data, you should consider the data types used and the most efficient placement of the variables.

USING EFFICIENT DATA TYPES

The data types you use should be considered carefully, because this can have a large impact on code size and code speed.

- Use `int` or `long` instead of `char` or `short` whenever possible, to avoid sign extension or zero extension. In particular, loop indexes should always be `int` or `long` to minimize code generation. Also, in Thumb mode, accesses through the stack pointer (`SP`) is restricted to 32-bit data types, which further emphasizes the benefits of using one of these data types.
- Use unsigned data types, unless your application really requires signed values.
- Be aware of the costs of using 64-bit data types, such as `double` and `long long`.

- Bitfields and packed structures generate large and slow code.
- Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

For information about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

In the compiler, the floating-point type `float` always uses the 32-bit format, and the type `double` always uses the 64-bit format.

Unless the application requires the extra precision that 64-bit floating-point numbers give, we recommend using 32-bit floating-point numbers instead.

By default, a *floating-point constant* in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, the `double` constant `1.0` is added and the result is converted back to a `float`:

```
double Test(float a)
{
    return a + 1.0;
}
```

To treat a floating-point constant as a `float` rather than as a `double`, add the suffix `f` to it, for example:

```
double Test(float a)
{
    return a + 1.0f;
}
```

For more information about floating-point types, see *Floating-point types*, page 294.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The ARM core requires that data in memory must be aligned. Each element in a structure must be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment correct.

There are situations when this can be a problem:

- There are external demands; for example, network communication protocols are usually specified in terms of data types with no padding in between
- You need to save data memory.

For information about alignment requirements, see *Alignment*, page 287.

There are two ways to solve the problem:

- Use the `#pragma pack` directive or the `__packed` data type attribute for a tighter layout of the structure. The drawback is that each access to an unaligned element in the structure will use more code.
- Write your own customized functions for *packing* and *unpacking* structures. This is a more portable way, which will not produce any more code apart from your functions. The drawback is the need for two views on the structure data—packed and unpacked.

For more information about the `#pragma pack` directive, see *pack*, page 328.

ANONYMOUS STRUCTS AND UNIONS

When a structure or union is declared without a name, it becomes anonymous. The effect is that its members will only be seen in the surrounding scope.

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the IAR C/C++ Compiler for ARM they can be used in C if language extensions are enabled.



In the IDE, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See *-e*, page 235, for additional information.

Example

In this example, the members in the anonymous union can be accessed, in function F, without explicitly specifying the union name:

```
struct S
{
    char mTag;
    union
    {
        long mL;
        float mF;
    };
} St;

void F(void)
{
    St.mL = 5;
}
```

The member names must be unique in the surrounding scope. Having an anonymous struct or union at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in this example:

```
__no_init volatile
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char way: 1;
        unsigned char out: 1;
    };
} @ 0x1000;

/* The variables are used here. */
void Test(void)
{
    IOPORT = 0;
    way = 1;
    out = 1;
}
```

This declares an I/O register byte IOPORT at address 0x1000. The I/O register has 2 bits declared, way and out. Note that both the inner structure and the outer union are anonymous.

Anonymous structures and unions are implemented in terms of objects named after the first field, with a prefix `_A_` to place the name in the implementation part of the namespace. In this example, the anonymous union will be implemented through an object named `_A_IOPORT`.

Controlling data and function placement in memory

The compiler provides different mechanisms for controlling placement of functions and data objects in memory. To use memory efficiently, you should be familiar with these mechanisms to know which one is best suited for different situations. You can use:

- The `@` operator and the `#pragma location` directive for absolute placement
 Use the `@` operator or the `#pragma location` directive to place individual global and static variables at absolute addresses. The variables must be declared `__no_init`.
 This is useful for individual data objects that must be located at a fixed address to conform to external requirements, for example to populate interrupt vectors or other hardware tables. Note that it is not possible to use this notation for absolute placement of individual functions.
- The `@` operator and the `#pragma location` directive for section placement
 Use the `@` operator or the `#pragma location` directive to place groups of functions or global and static variables in named sections, without having explicit control of each object. The sections can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the section begin and end operators. This is also useful if you want an interface between separately linked units, for example an application project and a boot loader project. Use named sections when absolute control over the placement of individual variables is not needed, or not useful.
- The `@` operator and the `#pragma location` directive for register placement
 Use the `@` operator or the `#pragma location` directive to place individual global and static variables in registers. The variables must be declared `__no_init`. This is useful for individual data objects that must be located in a specific register.
- The `--section` option
 Use the `--section` option to place functions and/or data objects in named sections, which is useful, for example, if you want to direct them to different fast or slow memories. For more information, see *--section*, page 255.

At compile time, data and functions are placed in different sections, see *Modules and sections*, page 68. At link time, one of the most important functions of the linker is to assign load addresses to the various sections used by the application. All sections, except for the sections holding absolute located data, are automatically allocated to memory according to the specifications in the linker configuration file, see *Placing code and data—the linker configuration file*, page 71.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

The @ operator, alternatively the #pragma location directive, can be used for placing global and static variables at absolute addresses.

To place a variable at an absolute address, the argument to the @ operator and the #pragma location directive should be a literal number, representing the actual address. The absolute location must fulfill the alignment requirement for the variable that should be located.

Note: All declarations of __no_init variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

Other variables placed at an absolute address use the normal distinction between declaration and definition. For these variables, you must provide the definition in only one module, normally with an initializer. Other modules can refer to the variable by using an extern declaration, with or without an explicit address.

Examples

In this example, a __no_init declared variable is placed at an absolute address. This is useful for interfacing between multiple processes, applications, etc:

```
__no_init volatile char alpha @ 0xFF2000; /* OK */
```

The next example contains two const declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. The next example contains a const declared object which is not initialized. The object is placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0xFF2002
__no_init const int beta;           /* OK */

const int gamma @ 0xFF2004 = 3;     /* OK */
```


In the first case, the value is not initialized by the compiler; the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

These examples show incorrect usage:

```
int delta @ 0xFF2006;                /* Error, not __no_init */
Error, neither */
/* "__no_init" nor "const".*/

__no_init int epsilon @ 0xFF2007;    /* Error, misaligned. */
```

C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```
volatile const __no_init int x @ 0x100;    /* Bad in C++ */
```

the linker will report that more than one variable is located at address 0x100.

To avoid this problem and make the process the same in C and C++, you should declare these variables `extern`, for example:

```
/* The extern keyword makes x public. */
extern volatile const __no_init int x @ 0x100;
```

Note: C++ static member variables can be placed at an absolute address just like any other static variable.

DATA AND FUNCTION PLACEMENT IN SECTIONS

The following methods can be used for placing data or functions in named sections other than default:

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named sections. The named section can either be a predefined section, or a user-defined section.
- The `--section` option can be used for placing variables and functions, which are parts of the whole compilation unit, in named sections.

C++ static member variables can be placed in named sections just like any other static variable.

If you use your own sections, in addition to the predefined sections, the sections must also be defined in the linker configuration file .

Note: Take care when explicitly placing a variable or function in a predefined section other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

The location of the sections can be controlled from the linker configuration file.

For more information about sections, see the chapter *Section reference*.

Examples of placing variables in named sections

In the following examples, a data object is placed in a user-defined section.

```
__no_init int alpha @ "MY_NOINIT"; /* OK */

#pragma location="MY_CONSTANTS"
const int beta; /* OK */
```

Examples of placing functions in named sections

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

#pragma location="MY_FUNCTIONS"
void h(void);
```

DATA PLACEMENT IN REGISTERS

The @ operator, alternatively the #pragma location directive, can be used for placing global and static variables in a register.

To place a variable in a register, the argument to the @ operator and the #pragma location directive should be an identifier that corresponds to an ARM core register in the range R4–R11 (R9 cannot be specified in combination with the --rwpi command line option).

A variable can be placed in a register only if it is declared as __no_init, has file scope, and its size is four bytes. A variable placed in a register does not have a memory address, so the address operator & cannot be used.

Within a module where a variable is placed in a register, the specified register will only be used for accessing that variable. The value of the variable is preserved across function calls to other modules because the registers R14–R11 are callee saved, and as such they

are restored when execution returns. However, the value of a variable placed in a register is not always preserved as expected:

- In an exception handler or library callback routine (such as the comparator function passed to `qsort`) the value might not be preserved. The value will be preserved if the command line option `--lock_regs` is used for locking the register in all modules of the application, including library modules.
- In a fast interrupt handler, the value of a variable in `R8–R11` is not preserved from outside the handler, because these registers are banked.
- The `longjmp` function and C++ exceptions might restore variables placed in registers to old values, unlike other variables with static storage duration which are not restored.

The linker does not prevent modules from placing different variables in the same register. Variables in different modules can be placed in the same register, and another module could use the register for other purposes.

Note: A variable placed in a register should be defined in an include file, to be included in every module that uses the variable. An unused definition in a module will cause the register to not be used in that module.

Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. See *optimize*, page 327, for information about the pragma directive.

MULTI-FILE COMPILATION UNITS

In addition to applying different optimizations to different source files or even functions, you can also decide what a compilation unit consists of—one or several source code files.

By default, a compilation unit consists of one source file, but you can also use multi-file compilation to make several source files in a compilation unit. The advantage is that interprocedural optimizations such as inlining, and cross jump have more source code to work on. Ideally, the whole application should be compiled as one compilation unit. However, for large applications this is not practical because of resource restrictions on the host computer. For more information, see *--mfc*, page 242.

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the optimizations to functions and variables that are actually used. For more information, see *--discard_unused_publics*, page 233.

OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. This table lists optimizations that are typically performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope
Medium	Same as above, and: Live-dead analysis and optimization Dead code elimination Redundant label elimination Redundant branch elimination Code hoisting Peephole optimization Some register content analysis and optimization Static clustering Common subexpression elimination

Table 25: Compiler optimization levels

Optimization level	Description
High (Balanced)	Same as above, and: Instruction scheduling Cross jumping Advanced register content analysis and optimization Loop unrolling Function inlining Code motion Type-based alias analysis

Table 25: Compiler optimization levels (Continued)

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 197.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--no_inline`, alternatively its equivalent in the IDE **Function inlining**, or the `#pragma optimize` directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis

- Static clustering
- Instruction scheduling.

Common subexpression elimination

Redundant re-evaluation of common subexpressions is by default eliminated at optimization levels Medium and High. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

Note: This option has no effect at optimization levels None and Low.

For more information about the command line option, see *--no_cse*, page 244.

Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most efficient for smaller loops, where the loop overhead can be a substantial part of the total loop body.

Loop unrolling, which can be performed at optimization level High, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler heuristically decides which loops to unroll. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

Note: This option has no effect at optimization levels None, Low, and Medium.

To disable loop unrolling, use the command line option *--no_unroll*, see *--no_unroll*, page 249.

Function inlining

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level High, normally reduces execution time, but the resulting code might be difficult to debug.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed. Normally, code size does not increase when optimizing for size. To control the heuristics for individual functions, use the `#pragma inline` directive or the Standard C `inline` keyword.

If you do not want to disable inlining for a whole module, use `#pragma inline=never` on an individual function instead.

Note: This option has no effect at optimization levels None, Low, and Medium.

For more information about the command line option, see *--no_inline*, page 245. For information about the pragma directive, see *inline*, page 324.

Code motion

Evaluation of loop-invariant expressions and common subexpressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization level High, normally reduces code size and execution time. The resulting code might however be difficult to debug.

Note: This option has no effect at optimization levels None, and Low.

For more information about the command line option, see *--no_code_motion*, page 243.

Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object are performed using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers can reference the same memory location or not.

Type-based alias analysis is performed at optimization level High. For application code conforming to standard C or C++ application code, this optimization can reduce code size and execution time. However, non-standard C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

Note: This option has no effect at optimization levels None, Low, and Medium.

For more information about the command line option, see *--no_tbaa*, page 248.

Example

```
short F(short *p1, long *p2)
{
    *p2 = 0;
    *p1 = 1;
    return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. If you use explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Static clustering

When static clustering is enabled, static and global variables that are defined within the same module are arranged so that variables that are accessed in the same function are stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses.

Note: This option has no effect at optimization levels None and Low.

For more information about the command line option, see *--no_clustering*, page 243.

Instruction scheduling

The compiler features an instruction scheduler to increase the performance of the generated code. To achieve that goal, the scheduler rearranges the instructions to minimize the number of pipeline stalls emanating from resource conflicts within the microprocessor. Note that not all cores benefit from scheduling. The resulting code might be difficult to debug.

Note: This option has no effect at optimization levels None, Low and Medium.

For more information about the command line option, see *--no_scheduling*, page 247.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code, for example:

- Using efficient addressing modes
- Helping the compiler optimize
- Generating more useful error message.

WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

The following is a list of programming techniques that will, when followed, enable the compiler to better optimize the application.

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called

functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.

- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables (non-static). Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 198. To maximize the effect of the inlining transformation, it is good practice to place the definitions of small functions called from more than one module in the header file rather than in the implementation file. Alternatively, you can use multi-file compilation. For more information, see *Multi-file compilation units*, page 196.
- If you need to access hardware resources, use SFRs or intrinsic functions if available. Otherwise, use inline assembler for short assembler sequences, otherwise use normal assembler routines. For more information, see *Mixing C and assembler*, page 129.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

FUNCTION PROTOTYPES

It is possible to declare and define functions using one of two different styles:

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. Using the prototyped style will also make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

To make the compiler verify that all functions have proper prototypes, use the compiler option **Require prototypes** (`--require_prototypes`).

Prototyped style

In prototyped function declarations, the type for each parameter must be specified.

```
int Test(char, int); /* Declaration */

int Test(char ch, int i) /* Definition */
{
    return i + ch;
}
```

Kernighan & Ritchie style

In K&R style—pre-Standard C—it is not possible to declare a function prototyped. Instead, an empty parameter list is used in the function declaration. Also, the definition looks different.

For example:

```
int Test();      /* Declaration */

int Test(ch, i) /* Definition */
char ch;
int i;
{
    return i + ch;
}
```

INTEGER TYPES AND BIT NEGATION

In some situations, the rules for integer types and their conversion lead to possibly confusing behavior. Things to look out for are assignments or conditionals (test expressions) involving types with different size, and logical operations, especially bit negation. Here, *types* also includes types of constants.

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant

conditionals. In this example an 8-bit character, a 32-bit integer, and two's complement is assumed:

```
void F1(unsigned char c1)
{
    if (c1 == ~0x80)
        ;
}
```

Here, the test is always false. On the right hand side, `0x80` is `0x00000080`, and `~0x00000080` becomes `0xFFFFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 24 bits set.

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 299.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several ARM devices are included in the IAR product installation. The header files are named `iodevice.h` and define the processor-specific special function registers (SFRs).

Note: Each header file contains one section used by the compiler, and one section used by the assembler.

SFRs with bitfields are declared in the header file. This example is from `ioks32c5000a.h`:

```
__no_init volatile union
{
    unsigned short mwctl2;
    struct
    {
        unsigned short edr : 1;
        unsigned short edw : 1;
    }
}
```

```

        unsigned short lee   : 2;
        unsigned short lemd  : 2;
        unsigned short lepl  : 2;
    } mwctl2bit;
} @ 0x1000;

/* By including the appropriate include file in your code,
 * it is possible to access either the whole register or any
 * individual bit (or bitfields) from C code as follows.
 */

void Test()
{
    /* Whole register access */
    mwctl2 = 0x1234;

    /* Bitfield accesses */
    mwctl2bit.edw = 1;
    mwctl2bit.lepl = 3;
}

```

You can also use the header files as templates when you create new header files for other ARM devices.

PASSING VALUES BETWEEN C AND ASSEMBLER OBJECTS

The following example shows how you in your C source code can use inline assembler to set and get values from a special purpose register:

```

#pragma diag_suppress=Pe940
#pragma optimize=no_inline
static unsigned long get_APSR( void )
{
    /* On function exit,
     * function return value should be present in R0 */
    asm( "MRS R0, APSR" );
}

#pragma diag_default=Pe940

#pragma optimize=no_inline
static void set_APSR( unsigned long value)
{
    /* On function entry, the first parameter is found in R0 */
    asm( "MSR APSR, R0" );
}

```

The general purpose register `R0` is used for getting and setting the value of the special purpose register `APSR`. As the functions only contain inline assembler, the compiler will not interfere with the register usage. The register `R0` is always used for return values. The first parameter is always passed in `R0` if the type is 32 bits or smaller.

The same method can be used also for accessing other special purpose registers and specific instructions.

To read more about inline assembler, see *Inline assembler*, page 131.

Note: Before you use inline assembler, see if you can use an intrinsic function instead. See *Summary of intrinsic functions*, page 337.

NON-INITIALIZED VARIABLES

Normally, the runtime environment will initialize all global and static variables when the application is started.

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in a separate section.

For `__no_init`, the `const` keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

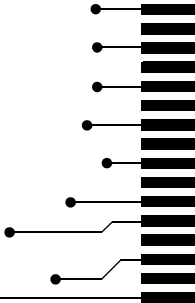
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For information about the `__no_init` keyword, see page 310. Note that to use this keyword, language extensions must be enabled; see *-e*, page 235. For information about the `#pragma object_attribute`, see page 327.

Part 2. Reference information

This part of the IAR C/C++ Development Guide for ARM® contains these chapters:

- External interface details
- Compiler options
- Linker options
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- The linker configuration file
- Section reference
- Stack usage control files
- IAR utilities
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89.





External interface details

This chapter provides reference information about how the compiler and linker interact with their environment. The chapter briefly lists and describes the invocation syntax, methods for passing options to the tools, environment variables, the include file search procedure, and finally the different types of compiler and linker output.

Invocation syntax

You can use the compiler and linker either from the IDE or from the command line. See the *IDE Project Management and Building Guide for ARM®* for information about using the build tools from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccam [options] [sourcefile] [options]
```

For example, when compiling the source file `prog.c`, use this command to generate an object file with debug information:

```
iccam prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order as they are specified on the command line.

If you run the compiler from the command line without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

ILINK INVOCATION SYNTAX

The invocation syntax for ILINK is:

```
ilinkarm [arguments]
```

Each argument is either a command-line option, an object file, or a library.

For example, when linking the object file `prog.o`, use this command:

```
ilinkarm prog.o --config configfile
```

If no filename extension is specified for the linker configuration file, the configuration file must have the extension `icf`.

Generally, the order of arguments on the command line is *not* significant. There is, however, one exception: when you supply several libraries, the libraries are searched in the same order that they are specified on the command line. The default libraries are always searched last.

The output executable image will be placed in a file named `a.out`, unless the `-o` option is used.

If you run `ILINK` from the command line without any arguments, the `ILINK` version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

PASSING OPTIONS

There are three different ways of passing options to the compiler and to `ILINK`:

- Directly from the command line
Specify the options on the command line after the `iccarm` or `ilinkarm` commands; see *Invocation syntax*, page 209.
- Via environment variables
The compiler and linker automatically append the value of the environment variables to every command line; see *Environment variables*, page 211.
- Via a text file, using the `-f` option; see *-f*, page 237.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *Compiler options* chapter.

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: C_INCLUDE=c:\program files\iar systems\embedded workbench 6.n\arm\inc;c:\headers
QCCARM	Specifies command line options; for example: QCCARM=-lA asm.lst

Table 26: Compiler environment variables

This environment variable can be used with ILINK:

Environment variable	Description
ILINKARM_CMD_LINE	Specifies command line options; for example: ILINKARM_CMD_LINE=--config full.icf --silent

Table 27: ILINK environment variables

Include file search procedure

This is a detailed description of the compiler’s #include file search procedure:

- If the name of the #include file is an absolute path specified in angle brackets or double quotes, that file is opened.
- If the compiler encounters the name of an #include file in angle brackets, such as:
#include <stdio.h>
it searches these directories for the file to include:
 - 1 The directories specified with the -I option, in the order that they were specified, see -I, page 239.
 - 2 The directories specified using the C_INCLUDE environment variable, if any; see *Environment variables*, page 211.
 - 3 The automatically set up library system include directories. See --dlib_config, page 234.
- If the compiler encounters the name of an #include file in double quotes, for example:
#include "vars.h"
it searches the directory of the source file in which the #include statement occurs, and then performs the same sequence as for angle-bracketed filenames.

If there are nested `#include` files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
#include "src.h"
...
src.h in directory dir\include
#include "config.h"
...
```

When `dir\exe` is the current directory, use this command for compilation:

```
icarm ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

<code>dir\include</code>	Current file is <code>src.h</code> .
<code>dir\src</code>	File including current file (<code>src.c</code>).
<code>dir\include</code>	As specified with the first <code>-I</code> option.
<code>dir\debugconfig</code>	As specified with the second <code>-I</code> option.

Use angle brackets for standard header files, like `stdio.h`, and double quotes for files that are part of your application.

For information about the syntax for including header files, see *Overview of the preprocessor*, page 367.

Compiler output

The compiler can produce the following output:

- A linkable object file
The object files produced by the compiler use the industry-standard format ELF. By default, the object file has the filename extension `.o`.
- Optional list files
Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 240. By default, these files will have the filename extension `.lst`.
- Optional preprocessor output files
A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `.i`.

- Diagnostic messages

Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 214.

- Error return codes

These codes provide status information to the operating system which can be tested in a batch file, see *Error return codes*, page 213.

- Size information

Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as *shared*.

Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

Error return codes

The compiler and linker return status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

Code	Description
0	Compilation or linking successful, but there might have been warnings.
1	Warnings were produced and the option <code>--warnings_affect_exit_code</code> was used.
2	Errors occurred.
3	Fatal errors occurred, making the tool abort.
4	Internal errors occurred, making the tool abort.

Table 28: Error return codes

ILINK output

ILINK can produce the following output:

- An absolute executable image

The final output produced by the IAR ILINK Linker is an absolute object file containing the executable image that can be put into an EPROM, downloaded to a hardware emulator, or executed on your PC using the IAR C-SPY Debugger Simulator. By default, the file has the filename extension `out`. The output format is always in ELF, which optionally includes debug information in the DWARF format.

- Optional logging information

During operation, ILINK logs its decisions on `stdout`, and optionally to a file. For example, if a library is searched, whether a required symbol is found in a library module, or whether a module will be part of the output. Timing information for each ILINK subsystem is also logged.

- Optional map files

A linker map file—containing summaries of linkage, runtime attributes, memory, and placement, as well as an entry list—can be generated by the ILINK option `--map`, see *--map*, page 276. By default, the map file has the filename extension `map`.

- Diagnostic messages

Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in the optional map file. For more information about diagnostic messages, see *Diagnostics*, page 214.

- Error return codes

ILINK returns status information to the operating system which can be tested in a batch file, see *Error return codes*, page 213.

- Size information about used memory and amount of time

Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` and displayed on the screen.

Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

MESSAGE FORMAT FOR THE COMPILER

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the compiler is produced in the form:

```
filename,linenumber level[tag]: message
```

with these elements:

<i>filename</i>	The name of the source file in which the issue was encountered
<i>linenumber</i>	The line number at which the compiler detected the issue
<i>level</i>	The level of seriousness of the issue
<i>tag</i>	A unique tag that identifies the diagnostic message
<i>message</i>	An explanation, possibly several lines long

Diagnostic messages are displayed on the screen, as well as printed in the optional list file.

Use the option `--diagnostics_tables` to list all possible compiler diagnostic messages.

MESSAGE FORMAT FOR THE LINKER

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from ILINK is produced in the form:

level[*tag*]: *message*

with these elements:

<i>level</i>	The level of seriousness of the issue
<i>tag</i>	A unique tag that identifies the diagnostic message
<i>message</i>	An explanation, possibly several lines long

Diagnostic messages are displayed on the screen, as well as printed in the optional map file.

Use the option `--diagnostics_tables` to list all possible linker diagnostic messages.

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler or linker finds a construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 254.

Warning

A diagnostic message that is produced when the compiler or linker finds a potential problem which is of concern, but which does not prevent completion of the compilation or linking. Warnings can be disabled by use of the command line option `--no_warnings`, see page 250.

Error

A diagnostic message that is produced when the compiler or linker finds a serious error. An error will produce a non-zero exit code.

Fatal error

A diagnostic message that is produced when the compiler finds a condition that not only prevents code generation, but which makes further processing pointless. After the message is issued, compilation terminates. A fatal error will produce a non-zero exit code.

SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all diagnostics messages, except for fatal errors and some of the regular errors.

See *Summary of compiler options*, page 222, for information about the compiler options that are available for setting severity levels.

For the compiler see also the chapter *Pragma directives*, for information about the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler or linker. It is produced using this form:

Internal error: *message*

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Include enough information to reproduce the problem, typically:

- The product name
- The version number of the compiler or of ILINK, which can be seen in the header of the list or map files generated by the compiler or by ILINK, respectively
- Your license number
- The exact internal error message text
- The files involved of the application that generated the internal error

- A list of the options that were used when the internal error occurred.

Compiler options

This chapter describes the syntax of compiler options and the general syntax rules for specifying option parameters, and gives detailed reference information about each option.

Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IDE.



See the *IDE Project Management and Building Guide for ARM®* for information about the compiler options available in the IDE and how to set them.

TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it can have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 210.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O or -Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac2004=n
```

Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

```
-I..\src or -I ..\src\
```

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

Rules for options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

For example, a short option with an optional parameter followed by a mandatory parameter:

```
-lA MyList.lst
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n PreprocOutput.lst
```

Rules for specifying a filename or directory as parameters

These rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally take a file path. The path can be relative or absolute. For example, to generate a listing to the file `List.lst` in the directory `..\listings\`:

```
iccarm prog.c -l ..\listings\List.lst
```

- For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name was specified with the option `-o`, in which case that name is used. For example:

```
iccarm prog.c -l ../listings\
```

The produced list file will have the default name `../listings\prog.lst`

- The *current directory* is specified with a period (`.`). For example:

```
iccarm prog.c -l .
```

- `/` can be used instead of `\` as the directory delimiter.
- By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

```
iccarm prog.c -l -
```

Additional rules

These rules also apply:

- When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead, you can prefix the parameter with two dashes; this example will create a list file called `-r`:

```
iccarm prog.c -l ---r
```

- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

```
--diag_warning=Be0001,Be0002
```

Alternatively, the option can be repeated for each argument, for example:

```
--diag_warning=Be0001
```

```
--diag_warning=Be0002
```

Summary of compiler options

This table summarizes the compiler command line options:

Command line option	Description
--aapcs	Specifies the calling convention
--aeabi	Enables AEABI-compliant code generation
--align_sp_on_irq	Generates code to align SP on entry to __irq functions
--arm	Sets the default function mode to ARM
--c89	Specifies the C89 dialect
--char_is_signed	Treats char as signed
--char_is_unsigned	Treats char as unsigned
--cpu	Specifies a processor variant
--c++	Specifies Standard C++
-D	Defines preprocessor symbols
--debug	Generates debug information
--dependencies	Lists file dependencies
--diag_error	Treats these as errors
--diag_remark	Treats these as remarks
--diag_suppress	Suppresses these diagnostics
--diag_warning	Treats these as warnings
--diagnostics_tables	Lists all diagnostic messages
--discard_unused_publics	Discards unused public symbols
--dlib_config	Uses the system include files for the DLIB library and determines which configuration of the library to use
-e	Enables language extensions
--ec++	Specifies Embedded C++
--eec++	Specifies Extended Embedded C++
--enable_hardware_workaround	Enables a specific hardware workaround
--enable_multibytes	Enables support for multibyte characters in source files
--endian	Specifies the byte order of the generated code and data

Table 29: Compiler options summary

Command line option	Description
<code>--enum_is_int</code>	Sets the minimum size on enumeration types
<code>--error_limit</code>	Specifies the allowed number of errors before compilation stops
<code>-f</code>	Extends the command line
<code>--fpu</code>	Selects the type of floating-point unit
<code>--guard_calls</code>	Enables guards for function static variable initialization
<code>--header_context</code>	Lists all referred source files and header files
<code>-I</code>	Specifies include file path
<code>--interwork</code>	Generates interworking code
<code>-l</code>	Creates a list file
<code>--legacy</code>	Generates object code linkable with older tool chains
<code>--lock_regs</code>	Prevents the compiler from using specified registers
<code>--macro_positions_in_diagnostics</code>	Obtains positions inside macros in diagnostic messages
<code>--mfc</code>	Enables multi-file compilation
<code>--migration_preprocessor_extensions</code>	Extends the preprocessor
<code>--misrac</code>	Enables error messages specific to MISRA-C:1998. This option is a synonym of <code>--misrac1998</code> and is only available for backwards compatibility.
<code>--misrac1998</code>	Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .
<code>--misrac2004</code>	Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .
<code>--misrac_verbose</code>	Enables verbose logging of MISRA C checking. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> or the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .
<code>--no_clustering</code>	Disables static clustering optimizations
<code>--no_code_motion</code>	Disables code motion optimization

Table 29: Compiler options summary (Continued)

Command line option	Description
<code>--no_const_align</code>	Disables the alignment optimization for constants.
<code>--no_cse</code>	Disables common subexpression elimination
<code>--no_exceptions</code>	Disables C++ exception support
<code>--no_fragments</code>	Disables section fragment handling
<code>--no_inline</code>	Disables function inlining
<code>--no_loop_align</code>	Disables alignment of labels in loops (Thumb2)
<code>--no_mem_idioms</code>	Disables idiom recognition for <code>memcpy/memset/memclr</code>
<code>--no_path_in_file_macros</code>	Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>
<code>--no_rtti</code>	Disables C++ RTTI support
<code>--no_rw_dynamic_init</code>	Disables runtime initialization of static C variables.
<code>--no_scheduling</code>	Disables the instruction scheduler
<code>--no_static_destruction</code>	Disables destruction of C++ static variables at program exit
<code>--no_system_include</code>	Disables the automatic search for system include files
<code>--no_tbaa</code>	Disables type-based alias analysis
<code>--no_typedefs_in_diagnostics</code>	Disables the use of typedef names in diagnostics
<code>--no_unaligned_access</code>	Avoids unaligned accesses
<code>--no_unroll</code>	Disables loop unrolling
<code>--no_warnings</code>	Disables all warnings
<code>--no_wrap_diagnostics</code>	Disables wrapping of diagnostic messages
<code>-O</code>	Sets the optimization level
<code>-o</code>	Sets the object filename. Alias for <code>--output</code> .
<code>--only_stdout</code>	Uses standard output only
<code>--output</code>	Sets the object filename
<code>--predef_macros</code>	Lists the predefined symbols.
<code>--preinclude</code>	Includes an include file before reading the source file
<code>--preprocess</code>	Generates preprocessor output
<code>--public_eqv</code>	Defines a global named assembler label
<code>-r</code>	Generates debug information. Alias for <code>--debug</code> .

Table 29: Compiler options summary (Continued)

Command line option	Description
--relaxed_fp	Relaxes the rules for optimizing floating-point expressions
--remarks	Enables remarks
--require_prototypes	Verifies that functions are declared before they are defined
--ropi	Generates code that uses PC-relative references to address code and read-only data.
--rwp_i	Generates code that uses an offset from the static base register to address-writable data.
--section	Changes a section name
--separate_cluster_for_initialized_variables	Separates initialized and non-initialized variables
--silent	Sets silent operation
--strict	Checks for strict compliance with Standard C/C++
--system_include_dir	Specifies the path for system include files
--thumb	Sets default function mode to Thumb
--use_c++_inline	Uses C++ inline semantics in C99
--use_unix_directory_separators	Uses / as directory separator in paths
--vla	Enables C99 VLA support
--warnings_affect_exit_code	Warnings affect exit code
--warnings_are_errors	Warnings are treated as errors

Table 29: Compiler options summary (Continued)


Descriptions of compiler options

The following section gives detailed reference information about each compiler option.




Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.


--aapcs

Syntax	<code>--aapcs={std vfp}</code>	
Parameters	<code>std</code>	Processor registers are used for floating-point parameters and return values in function calls according to standard AAPCS. <code>std</code> is the default when the software FPU is selected.
	<code>vfp</code>	VFP registers are used for floating-point parameters and return values. The generated code is not compatible with AEABI code. <code>vfp</code> is the default when a VFP unit is used.
Description	Use this option to specify the floating-point calling convention.	
		To set this option, use Project>Options>C/C++ Compiler>Extra Options .

--aeabi

Syntax	<code>--aeabi</code>	
Description	Use this option to generate AEABI-compliant object code. Note that this option must be used together with the <code>--guard_calls</code> option.	
See also	<i>AEABI compliance, page 182</i> and <i>--guard_calls, page 239</i> .	
		To set this option, use Project>Options>C/C++ Compiler>Extra Options .

--align_sp_on_irq

Syntax	<code>--align_sp_on_irq</code>	
Description	Use this option to align the stack pointer (SP) on entry to <code>__irq</code> declared functions. This is especially useful for nested interrupts, where the interrupted code uses the same SP as the interrupt handler. This means that the stack might only have 4-byte alignment, instead of the 8-byte alignment required by AEABI (and some instructions generated by the compiler for some cores).	
See also	<i>__irq, page 308</i> .	
		To set this option, use Project>Options>C/C++ Compiler>Extra Options .

--arm

Syntax	<code>--arm</code>
Description	Use this option to set default function mode to ARM. This setting must be the same for all files included in a program, unless they are interworking. Note: This option has the same effect as the <code>--cpu_mode=arm</code> option.
See also	<i>--interwork</i> , page 239 and <i>__interwork</i> , page 308.



Project>Options>C/C++ Compiler>Code>Processor mode>Arm

--c89

Syntax	<code>--c89</code>
Description	Use this option to enable the C89 C dialect instead of Standard C. Note: This option is mandatory when the MISRA C checking is enabled.
See also	<i>C language overview</i> , page 151.



Project>Options>C/C++ Compiler>Language 1>C dialect>C89

--char_is_signed

Syntax	<code>--char_is_signed</code>
Description	By default, the compiler interprets the plain <code>char</code> type as unsigned. Use this option to make the compiler interpret the plain <code>char</code> type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler. Note: The runtime library is compiled without the <code>--char_is_signed</code> option and cannot be used with code that is compiled with this option.



Project>Options>C/C++ Compiler>Language 2>Plain 'char' is

--char_is_unsigned

Syntax	--char_is_unsigned
Description	Use this option to make the compiler interpret the plain <code>char</code> type as unsigned. This is the default interpretation of the plain <code>char</code> type.



Project>Options>C/C++ Compiler>Language 2>Plain ‘char’ is

--cpu

Syntax	--cpu= <i>core</i>
Parameters	<i>core</i> Specifies a specific processor variant
Description	Use this option to select the processor variant for which the code is to be generated. The default is ARM7TDMI. The following cores and processor macrocells are recognized:

ARM7TDMI	ARM946E-S	ARM1176JF (alias for ARM1176JZF)
ARM7TDMI-S	ARM966E-S	ARM1176JF-S (alias for ARM1176JZF-S)
ARM710T	ARM968E-S	Cortex-A5
ARM720T	ARM10E	Cortex-M0
ARM740T	ARM1020E	Cortex-M1
ARM7EJ-S	ARM1022E	Cortex-Ms1*
ARM9TDMI	ARM1026EJ-S	Cortex-M3
ARM920T	ARM1136J	Cortex-M4
ARM922T	ARM1136J-S	Cortex-M4F
ARM940T	ARM1136JF	Cortex-R4
ARM9E	ARM1136JF-S	Cortex-R4F
ARM9E-S	ARM1176J (alias for ARM1176JZ)	XScale
ARM926EJ-S	ARM1176J-S (alias for ARM1176JZ-S)	XScale-IR7

* **Cortex-M1 with Operating System extension.**

See also

Processor variant, page 48.



Project>Options>General Options>Target>Processor configuration

--cpu_mode

Syntax

--cpu_mode={arm|a|thumb|t}

Parameters

arm, a (default)	Selects the arm mode as the default mode for functions
thumb, t	Selects the thumb mode as the default mode for functions

Description

Use this option to select the default mode for functions. This setting must be the same for all files included in a program, unless they are interworking.

See also

--interwork, page 239 and __interwork, page 308.



Project>Options>General Options>Target>Processor mode

--c++

Syntax

--c++

Description

By default, the language supported by the compiler is C. If you use Standard C++, you must use this option to set the language the compiler uses to C++.

See also

--ec++, page 235, --eec++, page 235, and Using C++, page 161.



Project>Options>C/C++ Compiler>Language 1>C++

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>C++

-D

Syntax	<code>-D symbol [=value]</code>	
Parameters	<i>symbol</i>	The name of the preprocessor symbol
	<i>value</i>	The value of the preprocessor symbol
Description	Use this option to define a preprocessor symbol. If no value is specified, 1 is used. This option can be used one or more times on the command line.	
	The option <code>-D</code> has the same effect as a <code>#define</code> statement at the top of the source file:	
	<code>-Dsymbol</code>	
	is equivalent to:	
	<code>#define symbol 1</code>	
	To get the equivalence of:	
	<code>#define FOO</code>	
	specify the <code>=</code> sign but nothing after, for example:	
	<code>-DFOO=</code>	



Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--debug, -r

Syntax	<code>--debug</code> <code>-r</code>	
Description	Use the <code>--debug</code> or <code>-r</code> option to make the compiler include information in the object modules required by the IAR C-SPY® Debugger and other symbolic debuggers.	
	Note: Including debug information will make the object files larger than otherwise.	



Project>Options>C/C++ Compiler>Output>Generate debug information

--dependencies

Syntax `--dependencies=[i|m] {filename|directory}`

Parameters

i (default)	Lists only the names of files
m	Lists in makefile style

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 220.

Description Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension `i`.

Example If `--dependencies` or `--dependencies=i` is used, the name of each opened input file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output is in makefile style. For each input file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of an input file. For example:

```
foo.o: c:\iar\product\include\stdio.h
foo.o: d:\myproject\include\foo.h
```

An example of using `--dependencies` with a popular make utility, such as `gmake` (GNU make):

- 1 Set up the rule for compiling files to be something like:

```
%.o : %.c
$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

That is, in addition to producing an object file, the command also produces a dependency file in makefile style (in this example, using the extension `.d`).

- 2 Include all the dependency files in the makefile using, for example:

```
-include $(sources:.c=.d)
```

Because of the dash (-) it works the first time, when the `.d` files do not yet exist.



This option is not available in the IDE.

--diag_error

Syntax	<code>--diag_error=tag[, tag, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number Pe117
Description	Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.	



Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors

--diag_remark

Syntax	<code>--diag_remark=tag[, tag, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number Pe177
Description	Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line. Note: By default, remarks are not displayed; use the <code>--remarks</code> option to display them.	



Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks

--diag_suppress

Syntax	<code>--diag_suppress=tag[, tag, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number Pe117

Description Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.



Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax `--diag_warning=tag[, tag, ...]`

Parameters

<i>tag</i>	The number of a diagnostic message, for example the message number Pe826
------------	--

Description Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.



Project>Options>C/C++ Compiler>Linker>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax `--diagnostics_tables {filename|directory}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters, page 220*.

Description Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.



This option is not available in the IDE.

--discard_unused_publics

Syntax `--discard_unused_publics`

Description Use this option to discard unused public functions and variables when compiling with the `--mfc` compiler option.

Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output.

See also *--mfc*, page 242 and *Multi-file compilation units*, page 196.



Project>Options>C/C++ Compiler>Discard unused publics

--dlib_config

Syntax `--dlib_config filename.h|config`

Parameters

<i>filename</i>	A DLIB configuration header file. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 220.
<i>config</i>	The default configuration file for the specified configuration will be used. Choose between: none, no configuration will be used normal, the normal library configuration will be used (default) full, the full library configuration will be used.

Description

Use this option to specify which library configuration to use, either by specifying an explicit file or by specifying a library configuration—in which case the default file for that library configuration will be used. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `arm\lib`. For examples and information about prebuilt runtime libraries, see *Using prebuilt libraries*, page 95.


If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see *Building and using a customized library*, page 105.




To set related options, choose:

Project>Options>General Options>Library Configuration


-e

Syntax	-e
Description	<p>In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.</p> <p>Note: The <code>-e</code> option and the <code>--strict</code> option cannot be used at the same time.</p>
See also	<p><i>Enabling language extensions, page 153.</i></p> <div>  <p>Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions</p> <p>Note: By default, this option is selected in the IDE.</p> </div>

--ec++

Syntax	--ec++
Description	<p>In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.</p> <div>  <p>Project>Options>C/C++ Compiler>Language 1>C++</p> <p>and</p> <p>Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++</p> </div>

--eec++

Syntax	--eec++
Description	<p>In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.</p>
See also	<p><i>Extended Embedded C++, page 162.</i></p> <div>  <p>Project>Options>C/C++ Compiler>Language 1>C++</p> <p>and</p> <p>Project>Options>C/C++ Compiler>Language 1>C++ dialect>Extended Embedded C++</p> </div>

--enable_hardware_workaround

Syntax	<code>--enable_hardware_workaround=<i>waid</i>[,<i>waid</i>[...]]</code>	
Parameters	<i>waid</i>	The ID number of the workaround to enable. For a list of available workarounds to enable, see the release notes.
Description	Use this option to make the compiler generate a workaround for a specific hardware problem.	
See also	The release notes for the compiler for a list of available parameters.	



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--enable_multibytes

Syntax	<code>--enable_multibytes</code>	
Description	By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support. Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.	



Project>Options>C/C++ Compiler>Language 2>Enable multibyte support

--endian

Syntax	<code>--endian={big b little l}</code>	
Parameters	<i>big</i> , <i>b</i>	Specifies big-endian as the default byte order
	<i>little</i> , <i>l</i> (default)	Specifies little-endian as the default byte order
Description	Use this option to specify the byte order of the generated code and data. By default, the compiler generates code in little-endian byte order.	

See also *Byte order, page 49* and *Byte order, page 288*.



Project>Options>General Options>Target>Endian mode

--enum_is_int

Syntax `--enum_is_int`

Description Use this option to force the size of all enumeration types to be at least 4 bytes.

Note: This option will not consider the fact that an `enum` type can be larger than an integer type.

See also *The enum type, page 289*.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--error_limit

Syntax `--error_limit=n`

Parameters

<code>n</code>	The number of errors before the compiler stops the compilation. <code>n</code> must be a positive integer; 0 indicates no limit.
----------------	---

Description Use the `--error_limit` option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.



This option is not available in the IDE.

-f

Syntax `-f filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters, page 220*.

Descriptions

Use this option to make the compiler read command line options from the named file, with the default filename extension `.xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--fpu

Syntax `--fpu={VFPv2 | VFPv3 | VFPv3_d16 | VFPv4 | VFPv4_sp | VFP9-S | none}`

Parameters

VFPv2	For a system that implements a VFP unit conforming to the architecture VFPv2.
VFPv3	For a system that implements a VFP unit conforming to the architecture VFPv3.
VFPv3_d16	For a system that implements a VFP unit conforming to the D16 variant of the architecture VFPv3.
VFPv4	For a system that implements a VFP unit conforming to the architecture VFPv4.
VFPv4_sp	For a system that implements a VFP unit conforming to the single-precision variant of the architecture VFPv4.
VFP9-S	VFP9-S is an implementation of the VFPv2 architecture that can be used with the ARM9E family of CPU cores. Selecting the VFP9-S coprocessor is therefore identical to selecting the VFPv2 architecture.
none (default)	The software floating-point library is used.

Description

Use this option to generate code that performs floating-point operations using a Vector Floating Point (VFP) coprocessor. By selecting a VFP coprocessor, you will override the use of the software floating-point library for all supported floating-point operations.

See also *VFP and floating-point arithmetic, page 49.*



Project>Options>General Options>Target>FPU

--guard_calls

Syntax	--guard_calls
Description	Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.
See also	<i>Managing a multithreaded environment, page 122.</i>



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--header_context

Syntax	--header_context
Description	Occasionally, to find the cause of a problem it is necessary to know which header file that was included from which source line. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.



This option is not available in the IDE.

-I

Syntax	-I <i>path</i>
Parameters	<i>path</i> The search path for #include files
Description	Use this option to specify the search paths for #include files. This option can be used more than once on the command line.
See also	<i>Include file search procedure, page 211.</i>



Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

--interwork

Syntax	--interwork
Description	Use this option to generate interworking code.

In code compiled with this option, functions will by default be of the type `interwork`. It will be possible to mix files compiled as `arm` and `thumb` (using the `--cpu_mode` option) as long as they are all compiled with the `--interwork` option.

Note: Source code compiled for an ARM architecture v5 or higher, or for AEABI compliance, is interworking by default.

See also `--interwork`, page 308.



Project>Options>General Options>Target>Generate interwork code

-1

Syntax `-1 [a|A|b|B|c|C|D] [N] [H] {filename|directory}`

Parameters

a (default)	Assembler list file
A	Assembler list file with C or C++ source as comments
b	Basic assembler list file. This file has the same contents as a list file produced with <code>-1a</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *
B	Basic assembler list file. This file has the same contents as a list file produced with <code>-1A</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
c	C or C++ list file
C (default)	C or C++ list file with assembler source as comments
D	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
N	No diagnostics in file
H	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

*** This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 220.

Description Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.



To set related options, choose:
Project>Options>C/C++ Compiler>List

--legacy

Syntax `--legacy={RVCT3.0}`

Parameters `RVCT3.0` Generates object code linkable with the linker in RVCT3.0. Use this mode together with the `--aeabi` option to export code that should be linked with the linker in RVCT3.0.

Description Use this option to generate code compatible with older tool chains.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**

--lock_regs

Syntax `--lock_regs=registers`

Parameters `registers` A comma-separated list of register names and register intervals to be locked, in the range R4–R11.


Description Use this option to prevent the compiler from generating code that uses the specified registers.

Example
`--lock_regs=R4`
`--lock_regs=R8–R11`
`--lock_regs=R4,R8–R11`



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**

--macro_positions_in_diagnostics

Syntax	<code>--macro_positions_in_diagnostics</code>
Description	Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.
	 To set this option, use Project>Options>C/C++ Compiler>Extra Options .

--mfc

Syntax	<code>--mfc</code>
Description	Use this option to enable <i>multi-file compilation</i> . This means that the compiler compiles one or several source files specified on the command line as one unit, which enhances interprocedural optimizations. Note: The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the <code>-o</code> compiler option and specify a certain output file.
Example	<code>iccarm myfile1.c myfile2.c myfile3.c --mfc</code>
See also	<code>--discard_unused_publics</code> , page 233, <code>--output</code> , <code>-o</code> , page 251, and <i>Multi-file compilation units</i> , page 196.



Project>Options>C/C++ Compiler>Multi-file compilation

--migration_preprocessor_extensions

Syntax	<code>--migration_preprocessor_extensions</code>
Description	If you need to migrate code from an earlier IAR Systems C or C++ compiler, you might want to use this option. Use this option to use the following in preprocessor expressions: <ul style="list-style-type: none">● Floating-point expressions● Basic type names and <code>sizeof</code>● All symbol names (including typedefs and variables). Note: If you use this option, not only will the compiler accept code that does not conform to Standard C, but it will also reject some code that <i>does</i> conform to the standard.

Important! Do not depend on these extensions in newly written code, because support for them might be removed in future compiler versions.



**Project>Options>C/C++ Compiler>Language 1>Enable IAR migration
preprocessor extensions**

--no_clustering

Syntax	<code>--no_clustering</code>
Description	Use this option to disable static clustering optimizations. Note: This option has no effect at optimization levels below Medium.
See also	<i>Static clustering, page 200.</i>



**Project>Options>C/C++ Compiler>Optimizations>Enable
transformations>Static clustering**

--no_code_motion

Syntax	<code>--no_code_motion</code>
Description	Use this option to disable code motion optimizations. Note: This option has no effect at optimization levels below Medium.
See also	<i>Code motion, page 199.</i>



**Project>Options>C/C++ Compiler>Optimizations>Enable
transformations>Code motion**

--no_const_align

Syntax	<code>--no_const_align</code>
Description	By default, the compiler uses alignment 4 for objects with a size of 4 bytes or more. Use this option to make the compiler align <code>const</code> objects based on the alignment of their type.

For example, a string literal will get alignment 1, because it is an array with elements of the type `const char` which has alignment 1. Using this option might save ROM space, possibly at the expense of processing speed.

See also *Alignment, page 287.*



To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--no_cse

Syntax `--no_cse`

Description Use this option to disable common subexpression elimination.
Note: This option has no effect at optimization levels below Medium.

See also *Common subexpression elimination, page 198.*



Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_exceptions

Syntax `--no_exceptions`

Description Use this option to disable exception support in the C++ language. Exception statements like `throw` and `try-catch`, and exception specifications on function definitions will generate an error message. Exception specifications on function declarations are ignored. The option is only valid when used together with the `--c++` compiler option.
If exceptions are not used in your application, it is recommended to disable support for them by using this option, because exceptions cause a rather large increase in code size.

See also *Exception handling, page 164.*



Project>Options>C/C++ Compiler>Language 1>C++
and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>C++>With exceptions

--no_fragments

Syntax `--no_fragments`

Description Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image. The effect of using this option in the compiler is smaller object size.

See also *Keeping symbols and sections, page 85.*



To set this option, use **Project>Options>C/C++ Compiler>Linker>Extra Options**

--no_inline

Syntax `--no_inline`

Description Use this option to disable function inlining.

Note: This option has no effect at optimization levels below High.

See also *Function inlining, page 198.*



Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_loop_align

Syntax `--no_loop_align`

Description Use this option to disable the 4-byte alignment of labels in loops. This option is only useful in Thumb2 mode.

In ARM/Thumb1 mode, this option is enabled but does not perform anything.

See also *Alignment, page 287.*



To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--no_mem_idioms

Syntax	<code>--no_mem_idioms</code>
Description	Use this option to make the compiler not optimize code sequences that clear, set, or copy a memory region. These memory access patterns (idioms) can otherwise be aggressively optimized, in some cases using calls to the runtime library. In principle, the transformation can involve more than a library call.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_path_in_file_macros

Syntax	<code>--no_path_in_file_macros</code>
Description	Use this option to exclude the path from the return value of the predefined preprocessor symbols <code>__FILE__</code> and <code>__BASE_FILE__</code> .
See also	<i>Descriptions of predefined preprocessor symbols, page 368.</i>



This option is not available in the IDE.

--no_rtti

Syntax	<code>--no_rtti</code>
Description	Use this option to disable the runtime type information (RTTI) support in the C++ language. RTTI statements like <code>dynamic_cast</code> and <code>typeid</code> will generate an error message. This option is only valid when used together with the <code>--c++</code> compiler option.
See also	<i>Using C++, page 161.</i>



Project>Options>C/C++ Compiler>Language 1>C++
and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>C++>With RTTI

--no_rw_dynamic_init

Syntax `--no_rw_dynamic_init`

Description Use this option to disable runtime initialization of static C variables.

C source code that is compiled with `--ropi` or `--rwpi` cannot have static pointer variables and constants initialized to addresses of objects that do not have a known address at link time. To solve this for writable static variables, the compiler generates code that performs the initialization at program startup (in the same way as dynamic initialization in C++).

See also `--ropi`, page 254 and `--rwpi`, page 255.



Project>Options>C/C++ Compiler>Code>No dynamic read/write initialization

--no_scheduling

Syntax `--no_scheduling`

Description Use this option to disable the instruction scheduler.

Note: This option has no effect at optimization levels below High.

See also *Instruction scheduling*, page 200.



Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling

--no_static_destruction

Syntax `--no_static_destruction`

Description Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.

Use this option to suppress the emission of such code.

See also *Setting up the atexit limit*, page 86.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_system_include

Syntax	<code>--no_system_include</code>
Description	By default, the compiler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the <code>-I</code> compiler option.
See also	<i>--dlib_config</i> , page 234, and <i>--system_include_dir</i> , page 257.



Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories

--no_tbaa

Syntax	<code>--no_tbaa</code>
Description	Use this option to disable type-based alias analysis. Note: This option has no effect at optimization levels below High.
See also	<i>Type-based alias analysis</i> , page 199.



Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis

--no_typedefs_in_diagnostics

Syntax	<code>--no_typedefs_in_diagnostics</code>
Description	Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.
Example	<pre>typedef int (*MyPtr)(char const *); MyPtr p = "foo";</pre> <p>will give an error message like this:</p> <pre>Error[Pe144]: a value of type "char *" cannot be used to initialize an entity of type "MyPtr"</pre>

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_unaligned_access

Syntax

`--no_unaligned_access`

Description

Use this option to make the compiler avoid unaligned accesses. Data accesses are usually performed aligned for improved performance. However, some accesses, most notably when reading from or writing to packed data structures, might be unaligned. When using this option, all such accesses will be performed using a smaller data size to avoid any unaligned accesses. This option is only useful for ARMv6 architectures and higher.

See also

Alignment, page 287.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_unroll

Syntax

`--no_unroll`

Description

Use this option to disable loop unrolling.

Note: This option has no effect at optimization levels below High.


See also

Loop unrolling, page 198.




Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax	--no_warnings
Description	By default, the compiler issues warning messages. Use this option to disable all warning messages.
	 This option is not available in the IDE.

--no_wrap_diagnostics

Syntax	--no_wrap_diagnostics
Description	By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.
	 This option is not available in the IDE.

-O

Syntax	-O [n l m h hs hz]	
Parameters	n	None* (Best debug support)
	l (default)	Low*
	m	Medium
	h	High, balanced
	hs	High, favoring speed
	hz	High, favoring size
	*The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.	
Description	Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only -O is used without any parameter, the optimization level High balanced is used. A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.	

See also

Controlling compiler optimizations, page 195.



Project>Options>C/C++ Compiler>Optimizations

--only_stdout

Syntax

`--only_stdout`

Description

Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).



This option is not available in the IDE.

--output, -o

Syntax

`--output {filename|directory}`
`-o {filename|directory}`

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters, page 220*.

Description

By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `o`. Use this option to explicitly specify a different output filename for the object code output.



This option is not available in the IDE.

--predef_macros

Syntax

`--predef_macros {filename|directory}`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters, page 220*.

Description

Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the `predef` filename extension.

Note that this option requires that you specify a source file on the command line.



This option is not available in the IDE.

--preinclude

Syntax

```
--preinclude includefile
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters, page 220*.

Description

Use this option to make the compiler read the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.



Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

--preprocess

Syntax

```
--preprocess [= [c] [n] [l]] {filename|directory}
```

Parameters

c	Preserve comments
n	Preprocess only
l	Generate <code>#line</code> directives

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters, page 220*.


Description

Use this option to generate preprocessed output to a named file.




Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

Syntax	<code>--public_equ symbol [=value]</code>	
Parameters	<i>symbol</i>	The name of the assembler symbol to be defined
	<i>value</i>	An optional value of the defined assembler symbol
Description	<p>This option is equivalent to defining a label in assembler language using the <code>EQU</code> directive and exporting it using the <code>PUBLIC</code> directive. This option can be used more than once on the command line.</p> <div>  This option is not available in the IDE. </div>	

--relaxed_fp

Syntax	<code>--relaxed_fp</code>	
Description	<p>Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:</p> <ul style="list-style-type: none"> • The expression consists of both single- and double-precision values • The double-precision values can be converted to single precision without loss of accuracy • The result of the expression is converted to single precision. <p>Note that performing the calculation in single precision instead of double precision might cause a loss of accuracy.</p>	
Example	<pre>float F(float a, float b) { return a + b * 3.0; }</pre> <p>The C standard states that <code>3.0</code> in this example has the type <code>double</code> and therefore the whole expression should be evaluated in <code>double</code> precision. However, when the <code>--relaxed_fp</code> option is used, <code>3.0</code> will be converted to <code>float</code> and the whole expression can be evaluated in <code>float</code> precision.</p> <div>  To set related options, choose: </div> <p>Project>Options>C/C++ Compiler>Language 2>Floating-point semantics</p>	

--remarks

Syntax	--remarks
Description	The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.
See also	<i>Severity levels, page 215.</i>



Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax	--require_prototypes
Description	<p>Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:</p> <ul style="list-style-type: none">• A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration• A function definition of a public function with no previous prototype declaration• An indirect function call through a function pointer with a type that does not include a prototype.



Project>Options>C/C++ Compiler>Language 1>Require prototypes

--ropi

Syntax	--ropi
Description	<p>Use this option to make the compiler generate code that uses PC-relative references to address code and read-only data.</p> <p>When this option is used, these limitations apply:</p> <ul style="list-style-type: none">• C++ constructions cannot be used• The object attribute <code>__ramfunc</code> cannot be used• Pointer constants cannot be initialized with the address of another constant, a string literal, or a function. However, writable variables can be initialized to constant addresses at runtime.

See also

--no_rw_dynamic_init, page 247, and *Descriptions of predefined preprocessor symbols*, page 368 for information about the preprocessor symbol `__ROPI__`.



Project>Options>C/C++ Compiler>Code>Code and read-only data (ropi)

--rwpi

Syntax

`--rwpi`

Description

Use this option to make the compiler generate code that uses the offset from the static base register (R9) to address-writable data.

When this option is used, these limitations apply:

- The object attribute `__ramfunc` cannot be used
- Pointer constants cannot be initialized with the address of a writable variable. However, static writable variables can be initialized to writable variable addresses at runtime.

See also

--no_rw_dynamic_init, page 247, and *Descriptions of predefined preprocessor symbols*, page 368 for information about the preprocessor symbol `__RWPI__`.



Project>Options>C/C++ Compiler>Code>Read/write data (rwpi)

--section

Syntax

`--section OldName=NewName`

Description

The compiler places functions and data objects into named sections which are referred to by the IAR ILINK Linker. Use this option to change the name of the section *OldName* to *NewName*.

This is useful if you want to place your code or data in different address ranges and you find the `@` notation, alternatively the `#pragma location` directive, insufficient. Note that any changes to the section names require corresponding modifications in the linker configuration file.

Example

To place functions in the section `MyText`, use:

```
--section .text=MyText
```

See also *Controlling data and function placement in memory, page 191.*



Project>Options>C/C++ Compiler>Output>Code section name

--separate_cluster_for_initialized_variables

Syntax	<code>--separate_cluster_for_initialized_variables</code>
Description	<p>Use this option to separate initialized and non-initialized variables when using variable clustering. This might reduce the number of bytes in the ROM area which are needed for data initialization, but it might lead to larger code.</p> <p>This option can be useful if you want to have your own data initialization routine, but want the IAR tools to arrange for the zero-initialized variables.</p>

See also *Manual initialization, page 87* and *Initialize directive, page 392.*



To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--silent

Syntax	<code>--silent</code>
Description	<p>By default, the compiler issues introductory messages and a final statistics report. Use this option to make the compiler operate without sending these messages to the standard output stream (normally the screen).</p> <p>This option does not affect the display of error and warning messages.</p>



This option is not available in the IDE.

--strict

Syntax	<code>--strict</code>
Description	<p>By default, the compiler accepts a relaxed superset of Standard C and C++. Use this option to ensure that the source code of your application instead conforms to strict Standard C and C++.</p>

Note: The `-e` option and the `--strict` option cannot be used at the same time.

See also *Enabling language extensions, page 153.*



Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict

--system_include_dir

Syntax `--system_include_dir path`

Parameters *path* The path to the system include files. For information about specifying a path, see *Rules for specifying a filename or directory as parameters, page 220.*

Description By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

See also *--dlib_config, page 234* and *--no_system_include, page 248.*



This option is not available in the IDE.

--thumb

Syntax `--thumb`

Description Use this option to set default function mode to Thumb. This setting must be the same for all files included in a program, unless they are interworking.

Note: This option has the same effect as the `--cpu_mode=thumb` option.

See also *--interwork, page 239* and *__interwork, page 308.*



Project>Options>C/C++ Compiler>Code>Processor mode>Thumb

--use_c++_inline

Syntax	<code>--use_c++_inline</code>
Description	<p>Standard C uses slightly different semantics for the <code>inline</code> keyword than C++ does. Use this option to get C++ semantics when compiling a Standard C source code file.</p> <p>The main difference in semantics is that in Standard C you cannot in general simply supply an inline definition in a header file. You need to supply an external definition in one of the compilation units, possibly by designating the inline definition as being external in that compilation unit.</p>



Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

--use_unix_directory_separators

Syntax	<code>--use_unix_directory_separators</code>
Description	<p>Use this option to make DWARF debug information use / (instead of \) as directory separators in file paths.</p> <p>This option can be useful if you have a debugger that requires directory separators in UNIX style.</p>



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.


--vla

Syntax	<code>--vla</code>
Description	<p>Use this option to enable support for C99 variable length arrays. This option requires Standard C and cannot be used together with the <code>--c89</code> compiler option.</p> <p>Note: <code>--vla</code> should not be used together with the <code>longjmp</code> library function, as that can lead to memory leakages.</p>
See also	<i>C language overview, page 151.</i>




Project>Options>C/C++ Compiler>Language 1>C dialect>Allow VLA

--warnings_affect_exit_code

Syntax	<code>--warnings_affect_exit_code</code>
Description	By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code.
	<div>  This option is not available in the IDE. </div>

--warnings_are_errors

Syntax	<code>--warnings_are_errors</code>
Description	<p>Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.</p> <p>Note: Any diagnostic messages that have been reclassified as warnings by the option <code>--diag_warning</code> or the <code>#pragma diag_warning</code> directive will also be treated as errors when <code>--warnings_are_errors</code> is used.</p>
See also	<code>--diag_warning</code> , page 233.
	<div>  Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors </div>

Linker options

This chapter gives detailed reference information about each linker option.

For general syntax rules, see *Options syntax*, page 219.

Summary of linker options

This table summarizes the linker options:

Command line option	Description
--basic_heap	Uses a basic heap instead of an advanced heap
--BE8	Uses the big-endian format BE8
--BE32	Uses the big-endian format BE32
--call_graph	Produces a call graph file in XML format
--config	Specifies the linker configuration file to be used by the linker
--config_def	Defines symbols for the configuration file
--cpp_init_routine	Specifies a user-defined C++ dynamic initialization routine
--cpu	Specifies a processor variant
--define_symbol	Defines symbols that can be used by the application
--dependencies	Lists file dependencies
--diag_error	Treats these message tags as errors
--diag_remark	Treats these message tags as remarks
--diag_suppress	Suppresses these diagnostic messages
--diag_warning	Treats these message tags as warnings
--diagnostics_tables	Lists all diagnostic messages
--enable_hardware_workaround	Enables specified hardware workaround
--enable_stack_usage	Enables stack usage analysis
--entry	Treats the symbol as a root symbol and as the start of the application
--error_limit	Specifies the allowed number of errors before linking stops

Table 30: Linker options summary

Command line option	Description
--exception_tables	Generates exception tables for C code
--export_builtin_config	Produces an icf file for the default configuration
--extra_init	Specifies an extra initialization routine that will be called if it is defined.
-f	Extends the command line
--force_exceptions	Always includes exception runtime code
--force_output	Produces an output file even if errors occurred
--image_input	Puts an image file in a section
--inline	Inlines small routines
--keep	Forces a symbol to be included in the application
--log	Enables log output for selected topics
--log_file	Directs the log to a file
--mangled_names_in_messages	Adds mangled names in messages
--map	Produces a map file
--merge_duplicate_sections	Merges equivalent read-only sections
--misrac1998	Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .
--misrac2004	Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .
--misrac_verbose	Enables verbose logging of MISRA C checking. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> and the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .
--no_dynamic_rtti_elimination	Includes dynamic runtime type information even when it is not needed.
--no_exceptions	Generates an error if exceptions are used
--no_fragments	Disables section fragment handling
--no_library_search	Disables automatic runtime library search
--no_locals	Removes local symbols from the ELF executable image.
--no_range_reservations	Disables range reservations for absolute symbols
--no_remove	Disables removal of unused sections

Table 30: Linker options summary (Continued)

Command line option	Description
--no_veneers	Disables generation of veneers
--no_vfe	Disables Virtual Function Elimination
--no_warnings	Disables generation of warnings
--no_wrap_diagnostics	Does not wrap long lines in diagnostic messages
-o	Sets the object filename. Alias for --output.
--only_stdout	Uses standard output only
--output	Sets the object filename
--pi_veneers	Generates position independent veneers.
--place_holder	Reserve a place in ROM to be filled by some other tool, for example a checksum calculated by ielftool.
--redirect	Redirects a reference to a symbol to another symbol
--remarks	Enables remarks
--search	Specifies more directories to search for object and library files
--semihosting	Links with debug interface
--silent	Sets silent operation
--skip_dynamic_initialization	Suppresses dynamic initialization at startup
--stack_usage_control	Specifies a stack usage control file
--strip	Removes debug information from the executable image
--vfe	Controls Virtual Function Elimination
--warnings_affect_exit_code	Warnings affects exit code
--warnings_are_errors	Warnings are treated as errors

Table 30: Linker options summary (Continued)

Descriptions of linker options

The following section gives detailed reference information about each compiler and linker option.



Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--basic_heap

Syntax

--basic_heap

Description

Use this option to use a basic heap instead of an advanced heap. The basic heap introduces less overhead and is suitable for example in applications that only allocate heap memory, but never calls `free`. For any other use of the heap, the advanced heap is usually more efficient.



To set this option, use **Project>Options>Linker>Extra Options**.

--BE8

Syntax

--BE8

Description

Use this option to specify the Byte Invariant Addressing mode.

This means that the linker reverses the byte order of the instructions, resulting in little-endian code and big-endian data. This is the default byte addressing mode for ARMv6 big-endian images. This is the only mode available for ARM v6M and ARM v7 with big-endian images.

Byte Invariant Addressing mode is only available on ARM processors that support ARMv6, ARM v6M, and ARM v7.

See also

Byte order, page 49, Byte order, page 288, --BE32, page 264, and --endian, page 236.



Project>Options>General Options>Target>Endian mode

--BE32

Syntax

--BE32

Description

Use this option to specify the legacy big-endian mode.

This produces big-endian code and data. This is the only byte-addressing mode for all big-endian images prior to ARMv6. This mode is also available for ARM v6 with big-endian, but not for ARM v6M or ARM v7.

See also

Byte order, page 49, Byte order, page 288, --BE8, page 264, and --endian, page 236.



Project>Options>General Options>Target>Endian mode

--call_graph

Syntax	<code>--call_graph {filename directory}</code>
Parameters	For information about specifying a filename or directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 220.
Description	Use this option to produce a call graph file. If no filename extension is specified, the extension <code>cgx</code> is used. This option can only be used once on the command line. Using this option enables stack usage analysis in the linker.
See also	<i>Stack usage analysis</i> , page 76



Project>Options>Linker>Advanced>Call graph output (XML)

--config

Syntax	<code>--config filename</code>
Parameters	For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 220.
Description	Use this option to specify the configuration file to be used by the linker (the default filename extension is <code>icf</code>). If no configuration file is specified, a default configuration is used. This option can only be used once on the command line.
See also	The chapter <i>The linker configuration file</i> .



Project>Options>Linker>Config>Linker configuration file

--config_def

Syntax	<code>--config_def symbol[=constant_value]</code>
Parameters	<div> <div><i>symbol</i></div> <div>The name of the symbol to be used in the configuration file. By default, the value 0 (zero) is used.</div> </div> <div> <div><i>constant_value</i></div> <div>The constant value of the configuration symbol.</div> </div>

Description	Use this option to define a constant configuration symbol to be used in the configuration file. This option has the same effect as the <code>define</code> symbol directive in the linker configuration file. This option can be used more than once on the command line.
See also	<i>--define_symbol</i> , page 267 and <i>Interaction between ILINK and the application</i> , page 89.



Project>Options>Linker>Config>Defined symbols for configuration file

--cpp_init_routine

Syntax	<code>--cpp_init_routine routine</code>
Parameters	<i>routine</i> A user-defined C++ dynamic initialization routine.
Description	<p>When using the IAR C/C++ compiler and the standard library, C++ dynamic initialization is handled automatically. In other cases you might need to use this option.</p> <p>If any sections with the section type <code>INIT_ARRAY</code> or <code>PREINIT_ARRAY</code> are included in your application, the C++ dynamic initialization routine is considered to be needed. By default, this routine is named <code>__iar_cstart_call_ctors</code> and is called by the startup code in the standard library. Use this option if you are not using the standard library and require another routine to handle these section types.</p>



To set this option, use **Project>Options>Linker>Extra Options**.

--cpu

Syntax	<code>--cpu=core</code>
Parameters	<i>core</i> Specifies a specific processor variant
Description	Use this option to select the processor variant for which the code is to be generated. The default is ARM7TDMI.
See also	<i>--cpu</i> , page 228 for a list of recognized cores and processor macrocells.



Project>Options>General Options>Target>Processor configuration

--define_symbol

Syntax	<code>--define_symbol symbol=constant_value</code>	
Parameters	<i>symbol</i>	The name of the constant symbol that can be used by the application.
	<i>constant_value</i>	The constant value of the symbol.
Description	Use this option to define a constant symbol, that is a label, that can be used by your application. This option can be used more than once on the command line. Note that his option is different from the <code>define symbol</code> directive.	
See also	<code>--config_def</code> , page 265 and <i>Interaction between ILINK and the application</i> , page 89.	



Project>Options>Linker>#define>Defined symbols

--dependencies

Syntax	<code>--dependencies=[i m] {filename directory}</code>	
Parameters	<i>i</i> (default)	Lists only the names of files
	<i>m</i>	Lists in makefile style
	For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 220.	
Description	Use this option to make the linker list the names of the linker configuration, object, and library files opened for input into a file with the default filename extension <i>i</i> .	
Example	<p>If <code>--dependencies</code> or <code>--dependencies=i</code> is used, the name of each opened input file, including the full path, if available, is output on a separate line. For example:</p> <pre>c:\myproject\foo.o d:\myproject\bar.o</pre>	

If `--dependencies=m` is used, the output is in makefile style. For each input file, one line containing a makefile dependency rule is produced. Each line consists of the name of the output file, a colon, a space, and the name of an input file. For example:

```
a.out: c:\myproject\foo.o
a.out: d:\myproject\bar.o
```



This option is not available in the IDE.

--diag_error

Syntax

```
--diag_error=tag[, tag, ...]
```

Parameters

<i>tag</i>	The number of a diagnostic message, for example the message number Pe117
------------	--

Description

Use this option to reclassify certain diagnostic messages as errors. An error indicates a problem of such severity that an executable image will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.



Project>Options>Linker>Diagnostics>Treat these as errors

--diag_remark

Syntax

```
--diag_remark=tag[, tag, ...]
```

Parameters

<i>tag</i>	The number of a diagnostic message, for example the message number Pe177
------------	--

Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a construction that may cause strange behavior in the executable image. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.



Project>Options>Linker>Diagnostics>Treat these as remarks

--diag_suppress

Syntax	<code>--diag_suppress=tag[, tag, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number <code>Pe117</code>
Description	Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.	



Project>Options>Linker>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax	<code>--diag_warning=tag[, tag, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number <code>Pe826</code>
Description	Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the linker to stop before linking is completed. This option may be used more than once on the command line.	



Project>Options>C/C++ CompilerLinker>Diagnostics>Treat these as warnings

--diagnostics_tables


Syntax	<code>--diagnostics_tables {filename directory}</code>	
Parameters	For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 220.	
Description	Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.	




This option cannot be given together with other options.

This option is not available in the IDE.

--enable_hardware_workaround

Syntax	<code>--enable_hardware_workaround=<i>waid</i>[,<i>waid</i>[...]]</code>	
Parameters	<i>waid</i>	The ID number of the workaround that you want to enable. For a list of available workarounds, see the release notes available in the Information Center.
Description	Use this option to make the linker generate a workaround for a specific hardware problem.	
See also	The release notes for the linker for a list of available parameters.	
		To set this option, use Project>Options>Linker>Extra Options .

--enable_stack_usage

Syntax	<code>--enable_stack_usage</code>	
Description	Use this option to enable stack usage analysis. If a linker map file is produced, a stack usage chapter is included in the map file. Note: If you use at least one of the <code>--stack_usage_control</code> or <code>--call_graph</code> options, stack usage analysis is automatically enabled.	
See also	<i>Stack usage analysis, page 76</i>	
		Project>Options>Linker>Advanced>Enable stack usage analysis

--entry

Syntax	<code>--entry <i>symbol</i></code>	
Parameters	<i>symbol</i>	The name of the symbol to be treated as a root symbol and start label

Description

Use this option to make a symbol be treated as a root symbol and the start label of the application. This is useful for loaders. If this option is not used, the default start symbol is `__iar_program_start`. A root symbol is kept whether or not it is referenced from the rest of the application, provided its module is included. A module in an object file is always included and a module part of a library is only included if needed.



Project>Options>Linker>Library>Override default program entry

--error_limit

Syntax `--error_limit=n`

Parameters

n The number of errors before the linker stops linking. *n* must be a positive integer; 0 indicates no limit.

Description

Use the `--error_limit` option to specify the number of errors allowed before the linker stops the linking. By default, 100 errors are allowed.



This option is not available in the IDE.

--exception_tables

Syntax `--exception_tables={nocreate|unwind|cantunwind}`

Parameters

<code>nocreate</code> (default)	Does not generate entries. Uses the least amount of memory, but the result is undefined if an exception is propagated through a function without exception information.
<code>unwind</code>	Generates unwind entries that enable an exception to be correctly propagated through functions without exception information.
<code>cantunwind</code>	Generates no-unwind entries so that any attempt to propagate an exception through the function will result in a call to <code>terminate</code> .

Description

Use this option to determine what the linker should do with functions that do not have exception information but which do have correct call frame information.

The compiler ensures that C functions get correct call frame information. For functions written in assembler language you need to use assembler directives to generate call frame information.

See also *Using C++, page 161.*



To set this option, use **Project>Options>Linker>Extra Options**.

--export_builtin_config

Syntax	<code>--export_builtin_config filename</code>
Parameters	For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters, page 220</i> .
Description	Exports the configuration used by default to a file.



This option is not available in the IDE.

--extra_init

Syntax	<code>--extra_init routine</code>		
Parameters	<table><tr><td><i>routine</i></td><td>A user-defined initialization routine.</td></tr></table>	<i>routine</i>	A user-defined initialization routine.
<i>routine</i>	A user-defined initialization routine.		
Description	Use this option to make the linker add an entry for the specified routine at the end of the initialization table. The routine will be called during system startup, after other initialization routines have been called and before <code>main</code> is called. Note that the routine must preserve the value passed to it in register <code>R0</code> . No entry is added if the routine is not defined.		



To set this option, use **Project>Options>Linker>Extra Options**.

-f

Syntax	<code>-f filename</code>
Parameters	For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters, page 220</i> .

Descriptions

Use this option to make the linker read command line options from the named file, with the default filename extension `.xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.



To set this option, use **Project>Options>Linker>Extra Options**.

--force_exceptions

Syntax

```
--force_exceptions
```

Description

Use this option to make the linker include exception tables and exception code even when the linker heuristics indicate that exceptions are not used.

The linker considers exceptions to be used if there is a `throw` expression that is not a `rethrow` in the included code. If there is no such `throw` expression in the rest of the code, the linker arranges for `operator new`, `dynamic_cast`, and `typeid` to call `abort` instead of throwing an exception on failure. If you need to catch exceptions from these constructs but your code contains no other throws, you might need to use this option.

See also

Using C++, page 161.



Project>Options>Linker>Optimizations>C++ Exceptions>Allow>Always include

--force_output

Syntax

```
--force_output
```

Description

Use this option to produce an output executable image regardless of any linking errors.



To set this option, use **Project>Options>Linker>Extra Options**

--image_input

Syntax	<code>--image_input filename [,symbol,[section[,alignment]]]</code>	
Parameters	<i>filename</i>	The pure binary file containing the raw image you want to link
	<i>symbol</i>	The symbol which the binary data can be referenced with.
	<i>section</i>	The section where the binary data will be placed; default is <code>.text</code> .
	<i>alignment</i>	The alignment of the section; default is 1.
Description	Use this option to link pure binary files in addition to the ordinary input files. The file's entire contents are placed in the section, which means it can only contain pure binary data.	
	The section where the contents of the <i>filename</i> file are placed, is only included if the symbol <i>symbol</i> is required by your application. Use the <code>--keep</code> option if you want to force a reference to the section.	
Example	<code>--image_input bootstrap.abs,Bootstrap,CSTARTUPCODE,4</code> The contents of the pure binary file <code>bootstrap.abs</code> are placed in the section <code>CSTARTUPCODE</code> . The section where the contents are placed is 4-byte aligned and will only be included if your application (or the command line option <code>--keep</code>) includes a reference to the symbol <code>Bootstrap</code> .	
See also	<code>--keep</code> , page 275.	



Project>Options>Linker>Input>Raw binary image

--inline

Syntax	<code>--inline</code>
Description	Some routines are so small that they can fit in the space of the instruction that calls the routine. Use this option to make the linker replace the call of a routine with the body of the routine, where applicable.



Project>Options>Linker>Optimizations>Inline small routines

--keep

Syntax	<code>--keep symbol</code>	
Parameters	<code>symbol</code>	The name of the symbol to be treated as a root symbol
Description	Normally, the linker keeps a symbol only if it is needed by your application. Use this option to make a symbol always be included in the final application.	



Project>Options>Linker>Input>Keep symbols

--log

Syntax	<code>--log topic[,topic,...]</code>	
Parameters	<code>topic</code> can be one of:	
	<code>initialization</code>	Lists copy batches and the compression selected for each batch.
	<code>libraries</code>	Lists all decisions taken by the automatic library selector. This might include extra symbols needed (<code>--keep</code>), redirections (<code>--redirect</code>), as well as which runtime libraries that were selected.
	<code>modules</code>	Lists each module that is selected for inclusion in the application, and which symbol that caused it to be included.
	<code>redirects</code>	Lists redirected symbols.
	<code>sections</code>	Lists each symbol and section fragment that is selected for inclusion in the application, and the dependence that caused it to be included.
	<code>unused_fragments</code>	Lists those section fragments that were not included in the application.
	<code>veeners</code>	Lists some veneer creation and usage statistics.
Description	Use this option to make the linker log information to <code>stdout</code> . The log information can be useful for understanding why an executable image became the way it is.	

See also `--log_file`, page 276.



Project>Options>Linker>List>Generate log

--log_file

Syntax	<code>--log_file filename</code>
Parameters	For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 220.
Description	Use this option to direct the log output to the specified file.
See also	<code>--log</code> , page 275.



Project>Options>Linker>List>Generate log

--mangled_names_in_messages

Syntax	<code>--mangled_names_in_messages</code>
Descriptions	Use this option to produce both mangled and unmangled names for C/C++ symbols in messages. Mangling is a technique used for mapping a complex C name or a C++ name (for example, for overloading) into a simple name. For example, <code>void h(int, char)</code> becomes <code>_Z1hic</code> .



This option is not available in the IDE.

--map

Syntax	<code>--map {filename directory}</code>
Description	<p>Use this option to produce a linker memory map file. The map file has the default filename extension <code>map</code>. The map file contains:</p> <ul style="list-style-type: none">● Linking summary in the map file header which lists the version of the linker, the current date and time, and the command line that was used.● Runtime attribute summary which lists runtime attributes.● Placement summary which lists each section/block in address order, sorted by placement directives.● Initialization table layout which lists the data ranges, packing methods, and compression ratios.● Module summary which lists contributions from each module to the image, sorted by directory and library.

- Entry list which lists all public and some local symbols in alphabetical order, indicating which module they came from.
- Some of the bytes might be reported as *shared*.
 Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

This option can only be used once on the command line.



Project>Options>Linker>List>Generate linker map file

--merge_duplicate_sections

Syntax	--merge_duplicate_sections
Description	Use this option to keep only one copy of equivalent read-only sections. Note that this can cause different functions or constants to have the same address, so an application that depends on the addresses being different will not work correctly with this option enabled.



Project>Options>Linker>Optimizations>Merge duplicate sections

--no_dynamic_rtti_elimination

Syntax	--no_dynamic_rtti_elimination
Description	<p>Use this option to make the linker include dynamic (polymorphic) runtime type information (RTTI) data in the application image even when the linker heuristics indicate that it is not needed.</p> <p>The linker considers dynamic runtime type information to be needed if there is a <code>typeid</code> or <code>dynamic_cast</code> expression for a polymorphic type in the included code. By default, if the linker detects no such expression, RTTI data will not be included just to make dynamic RTTI requests work.</p> <p>Note: A <code>typeid</code> expression for a <i>non</i>-polymorphic type results in a direct reference to a particular RTTI object and will not cause the linker to include any potentially unneeded objects.</p>

See also *Using C++, page 161.*



To set this option, use **Project>Options>Linker>Extra Options**.

--no_exceptions

Syntax `--no_exceptions`

Description Use this option to make the linker generate an error if there is a throw in the included code. This option is useful for making sure that your application does not use exceptions.

See also *Using C++, page 161.*



To set related options, choose:

Project>Options>Linker>Optimizations>Allow C++ exceptions

--no_fragments

Syntax `--no_fragments`

Description Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image. The effect of using this option in the compiler is smaller object size.

See also *Keeping symbols and sections, page 85.*



To set this option, use **Project>Options>C/C++ Compiler>Linker>Extra Options**

--no_library_search

Syntax `--no_library_search`

Description Use this option to disable the automatic runtime library search. This option turns off the automatic inclusion of the correct standard libraries. This is useful, for example, if the application needs a user-built standard library, etc.

Note that the option disables all steps of the automatic library selection, some of which might need to be reproduced if you are using the standard libraries. Use the `--log libraries` linker option together with automatic library selection enabled to determine which the steps are.



Project>Options>Linker>Library>Automatic runtime library selection

--no_locals

Syntax `--no_locals`

Description Use this option to remove local symbols from the ELF executable image.

Note: This option does not remove any local symbols from the DWARF information in the executable image.



Project>Options>Linker>Output

--no_range_reservations

Syntax `--no_range_reservations`

Description Normally, the linker reserves any ranges used by absolute symbols with a non-zero size, excluding them from consideration for `place in` commands.

When this option is used, these reservations are disabled, and the linker is free to place sections in such a way as to overlap the extent of absolute symbols.



To set this option, use **Project>Options>Linker>Extra Options**.

--no_remove

Syntax `--no_remove`

Description When this option is used, unused sections are not removed. In other words, each module that is included in the executable image contains all its original sections.

See also *Keeping symbols and sections, page 85.*



To set this option, use **Project>Options>Linker>Extra Options**.

--no_veneers

Syntax	<code>--no_veneers</code>
Description	Use this option to disable the insertion of veneers even though the executable image needs it. In this case, the linker will generate a relocation error for each reference that needs a veneer.
See also	<i>Veneers, page 90.</i>



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_vfe

Syntax	<code>--no_vfe</code>
Description	Use this option to disable the Virtual Function Elimination optimization. All virtual functions in all classes with at least one instance will be kept, and Runtime Type Information data will be kept for all polymorphic classes. Also, no warning message will be issued for modules that lack VFE information.
See also	<i>--vfe, page 285 and Virtual Function Elimination, page 181.</i>



To set related options, choose:
Project>Options>Linker>Advanced>PerformC++ Virtual Function Elimination

--no_warnings

Syntax	<code>--no_warnings</code>
Description	By default, the linker issues warning messages. Use this option to disable all warning messages.



This option is not available in the IDE.

--no_wrap_diagnostics

Syntax `--no_wrap_diagnostics`

Description By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.



This option is not available in the IDE.

--only_stdout

Syntax `--only_stdout`

Description Use this option to make the linker use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).



This option is not available in the IDE.

--output, -o

Syntax `--output {filename|directory}`
`-o {filename|directory}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 220.

Description By default, the object executable image produced by the linker is located in a file with the name `a.out`. Use this option to explicitly specify a different output filename, which by default will have the filename extension `out`.



Project>Options>Linker>Output>Output file

--pi_veneers

Syntax `--pi_veneers`

Description Use this option to make the linker generate position-independent veneers. Note that this type of veneer is larger and slower than normal veneers.

See also *Veneers, page 90.*



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--place_holder

Syntax `--place_holder symbol[,size[,section[,alignment]]]`

Parameters

<i>symbol</i>	The name of the symbol to create
<i>size</i>	Size in ROM; by default 4 bytes
<i>section</i>	Section name to use; by default <code>.text</code>
<i>alignment</i>	Alignment of section; by default 1

Description Use this option to reserve a place in ROM to be filled by some other tool, for example a checksum calculated by `ie1ftool`. Each use of this linker option results in a section with the specified name, size, and alignment. The symbol can be used by your application to refer to the section.

Note: Like any other section, sections created by the `--place_holder` option will only be included in your application if the section appears to be needed. The `--keep` linker option, or the `keep` linker directive can be used for forcing such section to be included.

See also *IAR utilities, page 417.*



To set this option, use **Project>Options>Linker>Extra Options**

--redirect

Syntax `--redirect from_symbol=to_symbol`

Parameters

<i>from_symbol</i>	The name of the source symbol
<i>to_symbol</i>	The name of the destination symbol

Description Use this option to change a reference from one symbol to another symbol.



To set this option, use **Project>Options>Linker>Extra Options**

--remarks

Syntax	<code>--remarks</code>
Description	The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the linker does not generate remarks. Use this option to make the linker generate remarks.
See also	<i>Severity levels, page 215.</i>



Project>Options>Linker>Diagnostics>Enable remarks

--search

Syntax	<code>--search path</code>
Parameters	<div> <div><code>path</code></div> <div>A path to a directory where the linker should search for object and library files.</div> </div>
Description	<p>Use this option to specify more directories for the linker to search for object and library files in.</p> <p>By default, the linker searches for object and library files only in the working directory. Each use of this option on the command line adds another search directory.</p>
See also	<i>The linking process, page 69.</i>



This option is not available in the IDE.

--semihosting

Syntax	<code>--semihosting[=iar_breakpoint]</code>
Parameters	<div> <div><code>iar_breakpoint</code></div> <div>The IAR-specific mechanism can be used when debugging applications that use SWI/SVC extensively.</div> </div>
Description	Use this option to include the debug interface—breakpoint mechanism—in the output image. If no parameter is specified, the SWI/SVC mechanism is included for ARM7/9/11, and the BKPT mechanism is included for Cortex-M.

See also *Application debug support, page 101.*



Project>Options>General Options>Library Configuration>Semihosted

--silent

Syntax `--silent`

Description By default, the linker issues introductory messages and a final statistics report. Use this option to make the linker operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.



This option is not available in the IDE.

--skip_dynamic_initialization

Syntax `--skip_dynamic_initialization`

Description Use this option to suppress dynamic initialization to be performed during system startup. Typically, this can be useful if you need to set up, for example, heap management for an RTOS before the initialization takes place.

In this case you must add a call to the library function `__iar_dynamic_initialization` in your application source code. Initialization will then take place at the time of the call to this function.



To set this option, use **Project>Options>Linker>Extra Options.**

--stack_usage_control

Syntax `--stack_usage_control filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters, page 220.*

Description Use this option to specify a stack usage control file. This file controls stack usage analysis, or provides more stack usage information for modules or functions. You can use this option multiple times to specify multiple stack usage control files. If no filename extension is specified, the extension `suc` is used.

Using this option enables stack usage analysis in the linker.

See also *Stack usage analysis, page 76*



Project>Options>Linker>Advanced>Control file

--strip

Syntax `--strip`

Description By default, the linker retains the debug information from the input object files in the output executable image. Use this option to remove that information.



To set related options, choose:

Project>Options>Linker>Output>Include debug information in output

--vfe

Syntax `--vfe [=forced]`

Parameters `forced` Performs Virtual Function Elimination even if one or more modules lack the needed virtual function elimination information.

Description Use this option to enable the Virtual Function Elimination optimization. By default, this optimization is enabled.

Forcing the use of Virtual Function Elimination can be unsafe if some of the modules that lack the needed information perform virtual function calls or use dynamic Runtime Type Information.


See also `--no_vfe`, page 280 and *Virtual Function Elimination, page 181*.




To set related options, choose:

Project>Options>Linker>Advanced>Perform C++ Virtual Function Elimination

--warnings_affect_exit_code

Syntax	--warnings_affect_exit_code
Description	By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code.
	 This option is not available in the IDE.

--warnings_are_errors

Syntax	--warnings_are_errors
Description	<p>Use this option to make the linker treat all warnings as errors. If the linker encounters an error, no executable image is generated. Warnings that have been changed into remarks are not treated as errors.</p> <p>Note: Any diagnostic messages that have been reclassified as warnings by the option <code>--diag_warning</code> will also be treated as errors when <code>--warnings_are_errors</code> is used.</p>
See also	<code>--diag_warning</code> , page 269 and <code>--diag_warning</code> , page 269.
	 Project>Options>Linker>Diagnostics>Treat all warnings as errors

Data representation

This chapter describes the data types, pointers, and structure types supported by the compiler.

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will have the same alignment as the structure member with the most strict alignment. To decrease the alignment requirements on the structure and its members, use `#pragma pack` or the `__packed` data type attribute.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure. For more information about pad bytes, see *Packed structure types*, page 298.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

ALIGNMENT ON THE ARM CORE

The alignment of a data object controls how it can be stored in memory. The reason for using alignment is that the ARM core can access 4-byte objects more efficiently when the object is stored at an address divisible by 4.

Objects with alignment 4 must be stored at an address divisible by 4, while objects with alignment 2 must be stored at addresses divisible by 2.

The compiler ensures this by assigning an alignment to every data type, ensuring that the ARM core will be able to read the data.

For related information, see *--align_sp_on_irq*, page 226 and *--no_const_align*, page 243.

Byte order

The ARM core stores data in either little-endian or big-endian byte order. To specify the byte order, use the *--endian* compiler option; see *--endian*, page 236.

In the little-endian byte order, which is default, the *least* significant byte is stored at the lowest address in memory. The *most* significant byte is stored at the highest address.

In the big-endian byte order, the *most* significant byte is stored at the lowest address in memory. The *least* significant byte is stored at the highest address. If you use the big-endian byte order, it might be necessary to use the `#pragma bitfields=reversed` directive to be compatible with code for other compilers and I/O register definitions of some devices, see *Bitfields*, page 290.

Note: There are two variants of the big-endian mode, BE8 and BE32, which you specify at link time. In BE8 data is big-endian and code is little-endian. In BE32 both data and code are big-endian. In architectures before v6, the BE32 endian mode is used, and after v6 the BE8 mode is used. In the v6 (ARM11) architecture, both big-endian modes are supported.

Basic data types

The compiler supports both all Standard C basic data types and some additional types.

INTEGER TYPES

This table gives the size and range of each integer data type:

Data type	Size	Range	Alignment
bool	8 bits	0 to 1	1
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1

Table 31: Integer types

Data type	Size	Range	Alignment
signed short	16 bits	-32768 to 32767	2
unsigned short	16 bits	0 to 65535	2
signed int	32 bits	-2^{31} to $2^{31}-1$	4
unsigned int	32 bits	-2^{31} to $2^{31}-1$	4
signed long	32 bits	-2^{31} to $2^{31}-1$	4
unsigned long	32 bits	0 to $2^{32}-1$	4
signed long long	64 bits	-2^{63} to $2^{63}-1$	8
unsigned long long	64 bits	0 to $2^{64}-1$	8

Table 31: Integer types (Continued)

Signed variables are represented using the two’s complement form.

Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring signed rather than unsigned.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example:

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

For related information, see *--enum_is_int*, page 237.

The char type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

The `wchar_t` type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

Bitfields

In Standard C, `int`, `signed int`, and `unsigned int` can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (`char`, `short`, `int`, etc) results in a signed or unsigned bitfield.

In the IAR C/C++ Compiler for ARM, plain integer types are treated as unsigned.

Bitfields in expressions are treated as `int` if `int` can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.

Each bitfield is placed in the next container of its base type that has enough available bits to accommodate the bitfield. Within each container, the bitfield is placed in the first available byte or bytes, taking the byte order into account.

In addition, the compiler supports an alternative bitfield allocation strategy (disjoint types), where bitfield containers of different types are not allowed to overlap. Using this allocation strategy, each bitfield is placed in a new container if its type is different from that of the previous bitfield, or if the bitfield does not fit in the same container as the previous bitfield. Within each container, the bitfield is placed from the least significant bit to the most significant bit (disjoint types) or from the most significant bit to the least significant bit (reverse disjoint types). This allocation strategy will never use less space than the default allocation strategy (joined types), and can use significantly more space when mixing bitfield types.

Use the `#pragma bitfield` directive to choose which bitfield allocation strategy to use, see *bitfields*, page 319.

Example

Assume this example:

```
struct BitfieldExample
{
    uint32_t a : 12;
    uint16_t b : 3;
    uint16_t c : 7;
    uint8_t d;
};
```

The example in the joined types bitfield allocation strategy

To place the first bitfield, *a*, the compiler allocates a 32-bit container at offset 0 and puts *a* into the first and second bytes of the container.

For the second bitfield, *b*, a 16-bit container is needed and because there are still four bits free at offset 0, the bitfield is placed there.

For the third bitfield, *c*, as there is now only one bit left in the first 16-bit container, a new container is allocated at offset 2, and *c* is placed in the first byte of this container.

The fourth member, *d*, can be placed in the next available full byte, which is the byte at offset 3.

In little-endian mode, each bitfield is allocated starting from the least significant free bit of its container to ensure that it is placed into bytes from left to right.

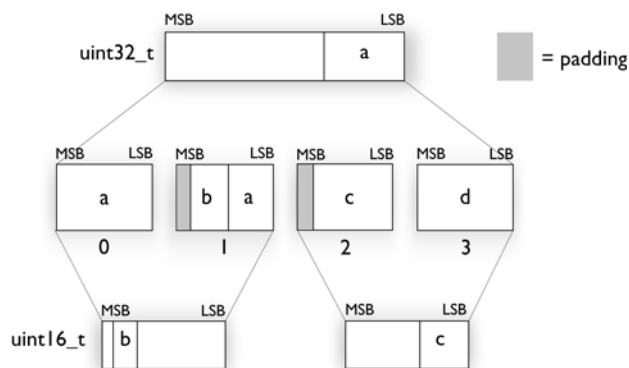


Figure 14: Layout of bitfield members for joined types in little-endian mode

In big-endian mode, each bitfield is allocated starting from the most significant free bit of its container to ensure that it is placed into bytes from left to right.

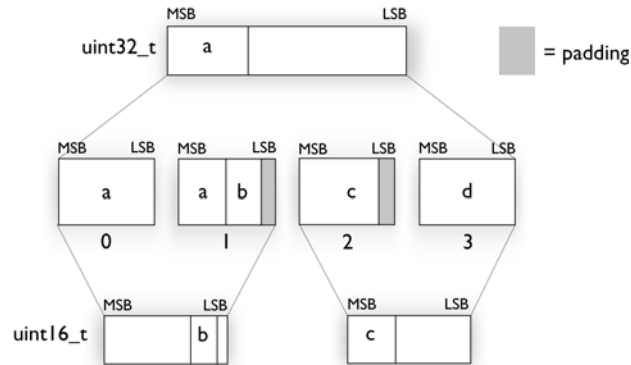


Figure 15: Layout of bitfield members for joined types in big-endian mode

The example in the disjoint types bitfield allocation strategy

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset 0 and puts `a` into the least significant 12 bits of the container.

To place the second bitfield, `b`, a new container is allocated at offset 4, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

The fourth member, `d`, is allocated into the byte at offset 6. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order (reverse disjoint types), each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example` in little-endian mode:

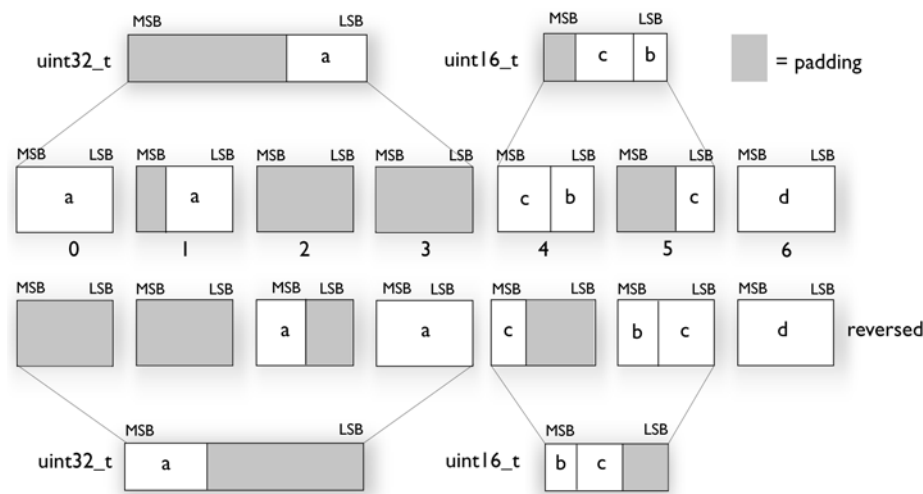


Figure 16: Layout of `bitfield_example` for disjoint types in little-endian mode

This is the layout of `bitfield_example` in big-endian mode:

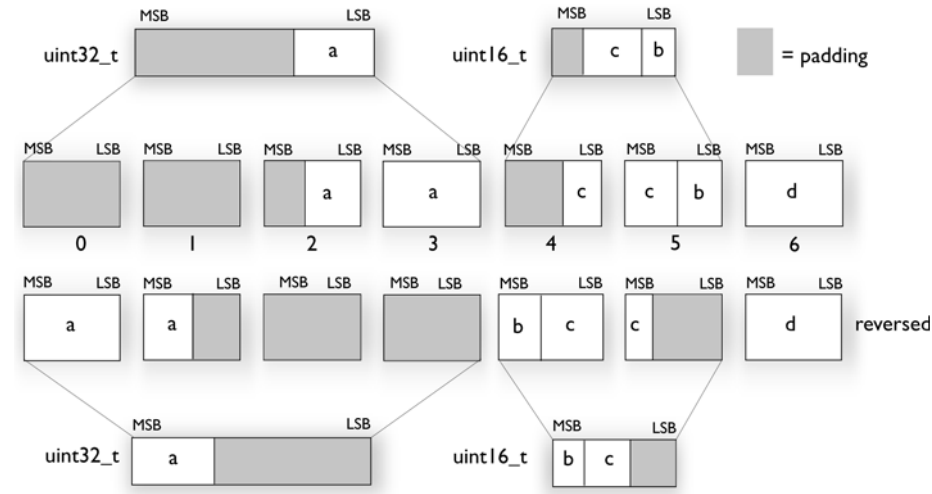


Figure 17: Layout of `bitfield_example` for disjoint types in big-endian mode

FLOATING-POINT TYPES

In the IAR C/C++ Compiler for ARM, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

Type	Size	Range (+/-)	Decimals	Exponent	Mantissa	Alignment
float	32 bits	±1.18E-38 to ±3.40E+38	7	8 bits	23 bits	4
double	64 bits	±2.23E-308 to ±1.79E+308	15	11 bits	52 bits	8
long double	64 bits	±2.23E-308 to ±1.79E+308	15	11 bits	52 bits	8

Table 32: Floating-point types

For Cortex-M0 and Cortex-M1, the compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero. For information about the representation of subnormal numbers for other cores, see *Representation of special floating-point numbers*, page 295.

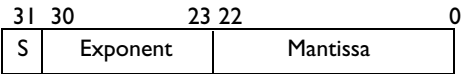
Floating-point environment

Exception flags for floating-point values are supported for devices with a VFP unit, and they are defined in the `fenv.h` file. For devices without a VFP unit, the functions defined in the `fenv.h` file exists but have no functionality.

The `feraiseexcept` function does not raise an `inexact` floating-point exception when called with `FE_OVERFLOW` or `FE_UNDERFLOW`.

32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:



The exponent is 8 bits, and the mantissa is 23 bits.

The value of the number is:

$(-1)^S * 2^{(Exponent-127)} * 1.Mantissa$

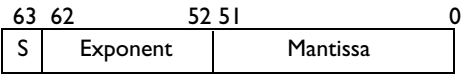
The range of the number is at least:

$\pm 1.18E-38 \text{ to } \pm 3.39E+38$

The precision of the float operators (+, -, *, and /) is approximately 7 decimal digits.

64-bit floating-point format

The representation of a 64-bit floating-point number as an integer is:



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

$(-1)^S * 2^{(Exponent-1023)} * 1.Mantissa$

The range of the number is at least:

$\pm 2.23E-308$ to $\pm 1.79E+308$

The precision of the float operators (+, -, *, and /) is approximately 15 decimal digits.

Representation of special floating-point numbers

This list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and at least one bit set in the 20 most significant bits of the mantissa. Remaining bits are zero.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and the most significant bit in the mantissa to 1. The value of the sign bit is ignored.
- Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is denormalized, even though the number is treated as if the exponent was 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a denormalized number is:

$(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$

where BIAS is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

The size of function pointers is always 32 bits and the range is 0x0–0xFFFFFFFF.

When function pointer types are declared, attributes are inserted before the * sign, for example:

```
typedef void (__thumb __interwork * IntHandler) (void);
```

This can be rewritten using #pragma directives:

```
#pragma type_attribute=__thumb __interwork
typedef void IntHandler_function(void);
typedef IntHandler_function *IntHandler;
```

DATA POINTERS

There is one data pointer available. Its size is 32 bits and the range is 0x0–0xFFFFFFFF.

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an unsigned integer type to a pointer of a larger type is performed by zero extension
- Casting a *value* of an unsigned integer type to a pointer of a larger type is performed by sign extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the IAR C/C++ Compiler for ARM, the size of `size_t` is 32 bits.

ptrdiff_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the IAR C/C++ Compiler for ARM, the size of `ptrdiff_t` is 32 bits.

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for ARM, the size of `intptr_t` is 32 bits.

uintptr_t

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned.

Structure types

The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

ALIGNMENT

The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. The size of a `struct` is also adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

Example

```
struct First
{
    char c;
    short s;
} s;
```

This diagram shows the layout in memory:

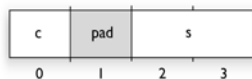


Figure 18: Structure layout

The alignment of the structure is 2 bytes, and a pad byte must be inserted to give `short s` the correct alignment.

PACKED STRUCTURE TYPES

The `__packed` data type attribute or the `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work, because the normal code might depend on the alignment being correct.

Example

This example declares a packed structure:

```
#pragma pack(1)
struct S
{
    char c;
    short s;
};

#pragma pack()
```

In this example, the structure `S` has this memory layout:

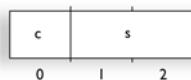


Figure 19: Packed structure layout

This example declares a new non-packed structure, `S2`, that contains the structure `s` declared in the previous example:

```
struct S2
{
    struct S s;
    long l;
};
```

`S2` has this memory layout

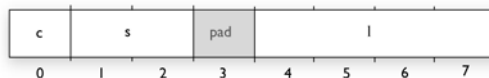


Figure 20: Packed structure layout

The structure `s` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 2, which means that alignment of the structure `S2` will become 2.

For more information, see *Alignment of elements in a structure*, page 189.

Type qualifiers

According to the C standard, `volatile` and `const` are type qualifiers.

DECLARING OBJECTS VOLATILE

By declaring an object `volatile`, the compiler is informed that the value of the object can change beyond the compiler's control. The compiler must also assume that any accesses can have side effects—thus all accesses to the `volatile` object must be preserved.

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The C standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine:

- The compiler considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5;    /* A write access */
a += 6;   /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type
- Adding a `const` qualifier to a `volatile` object will make write accesses to the object impossible. However, the object will be placed in RAM as specified by the C standard.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for ARM are described below.

Rules for accesses

In the IAR C/C++ Compiler for ARM, accesses to `volatile` declared objects are subject to these rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The compiler adheres to these rules for accesses to all 8-, 16-, and 32-bit scalar types, except for accesses to unaligned 16- and 32-bit fields in packed structures.

For all combinations of object types not listed, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, define the variable like this:

```
const volatile int x @ "FLASH";
```

The compiler will generate the read/write section `FLASH`. That section should be placed in ROM and is used for manually initializing the variables when the application starts up.

Thereafter, the initializers can be reflashed with other values at any time.

DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` are allocated in ROM.

In C++, objects that require runtime initialization cannot be placed in ROM.

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.

Extended keywords

This chapter describes the extended keywords that support specific features of the ARM core and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The compiler provides a set of attributes that can be used on functions or data objects to support specific features of the ARM core. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For more information about each attribute, see *Descriptions of extended keywords*, page 307.

Note: The extended keywords are only available when language extensions are enabled in the compiler.



In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 249 for more information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes explicitly in your declarations, or use the pragma directive `#pragma type_attribute`.

General type attributes

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__arm`, `__fiq`, `__interwork`, `__irq`, `__swi`, `__task`, and `__thumb`
- *Data type attributes*: `__big_endian`, `const`, `__little_endian`, `__packed`, and `volatile`.

You can specify as many type attributes as required for each level of pointer indirection.

For more information about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 313.

Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the `__little_endian` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` will be accessed with little endian byte order. However, note that an individual member of a `struct` or `union` cannot have a type attribute. The variables `k` and `l` behave in the same way:

```
__little_endian int i, j;
int __little_endian k, l;
```

Note that the attribute affects both identifiers.

This declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__little_endian
int i, j;
```

The advantage of using `pragma` directives for specifying keywords is that it offers you a method to make sure that the source code is portable.

Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

<code>int __packed * p;</code>	A pointer to a packed integer.
<code>int * __packed p;</code>	A packed pointer to an integer.
<code>__packed int * p;</code>	A packed pointer to an integer.

Syntax for type attributes on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```
__irq __arm void my_handler(void);
```

or

```
void (__irq __arm my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__irq __arm
void my_handler(void);
```

OBJECT ATTRIBUTES

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables: `__absolute`, `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, `__root`, and `__weak`
- Object attributes that can be used for functions: `__intrinsic`, `__nested`, `__noreturn`, `__ramfunc`, and `__stackless`.

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 205.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. This declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: Object attributes cannot be used in combination with the `typedef` keyword.

Summary of extended keywords

This table summarizes the extended keywords:

Extended keyword	Description
<code>__absolute</code>	Makes references to the object use absolute addressing
<code>__arm</code>	Makes a function execute in ARM mode
<code>__big_endian</code>	Declares a variable to use the big-endian byte order
<code>__fiq</code>	Declares a fast interrupt function
<code>__interwork</code>	Declares a function to be callable from both ARM and Thumb mode
<code>__intrinsic</code>	Reserved for compiler internal use only
<code>__irq</code>	Declares an interrupt function
<code>__little_endian</code>	Declares a variable to use the little-endian byte order
<code>__nested</code>	Allows an <code>__irq</code> declared interrupt function to be nested, that is, interruptible by the same type of interrupt
<code>__no_init</code>	Supports non-volatile memory
<code>__noreturn</code>	Informs the compiler that the function will not return
<code>__packed</code>	Decreases data type alignment to 1
<code>__pcrel</code>	Used internally by the compiler for constant data when the <code>--ropi</code> compiler option is used
<code>__ramfunc</code>	Makes a function execute in RAM
<code>__root</code>	Ensures that a function or variable is included in the object code even if unused
<code>__sbrel</code>	Used internally by the compiler for constant data when the <code>--rwpi</code> compiler option is used
<code>__stackless</code>	Makes a function callable without a working stack
<code>__swi</code>	Declares a software interrupt function
<code>__task</code>	Relaxes the rules for preserving registers
<code>__thumb</code>	Makes a function execute in Thumb mode
<code>__weak</code>	Declares a symbol to be externally weakly linked

Table 33: Extended keywords summary

Descriptions of extended keywords

These sections give detailed information about each extended keyword.

__absolute

Syntax	Follows the generic syntax rules for object attributes that can be used on data, see <i>Object attributes</i> , page 305.
Description	<p>The <code>__absolute</code> keyword makes references to the object use absolute addressing.</p> <p>The following limitations apply:</p> <ul style="list-style-type: none"> ● Only available when the <code>--ropi</code> or <code>--rwpi</code> compiler option is used ● Can only be used on external declarations.
Example	<pre>extern __absolute char otherBuffer[100];</pre>

__arm

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 303.
Description	<p>The <code>__arm</code> keyword makes a function execute in ARM mode. An <code>__arm</code> declared function can, unless it is also declared <code>__interwork</code>, only be called from functions that also execute in ARM mode.</p> <p>A function declared <code>__arm</code> cannot be declared <code>__thumb</code>.</p> <p>Note: Non-interwork ARM functions cannot be called from Thumb mode.</p>
Example	<pre>__arm int func1(void);</pre>
See also	<code>__interwork</code> , page 308.

__big_endian

Syntax	Follows the generic syntax rules for type attributes that can be used on data objects, see <i>Type attributes</i> , page 303.
Description	<p>The <code>__big_endian</code> keyword is used for accessing a variable that is stored in the big-endian byte order regardless of what byte order the rest of the application uses. The <code>__big_endian</code> keyword is available when you compile for ARMv6 or higher.</p>

Note that this keyword cannot be used on pointers. Also, this attribute cannot be used on arrays.

Example `__big_endian long my_variable;`

See also `__little_endian`, page 309.

__fiq

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 303.

Description The `__fiq` keyword declares a fast interrupt function. All interrupt functions must be compiled in ARM mode. A function declared `__fiq` does not accept parameters and does not have a return value. This keyword is not available when you compile for Cortex-M devices.

Example `__fiq __arm void interrupt_function(void);`

__interwork

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 303.

Description A function declared `__interwork` can be called from functions executing in either ARM or Thumb mode.

Note: By default, functions are interwork when the `--interwork` compiler option is used, and when the `--cpu` option is used and it specifies a core where interwork is default.

Example `typedef void (__thumb __interwork *IntHandler)(void);`

__intrinsic

Description The `__intrinsic` keyword is reserved for compiler internal use only.

__irq

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 303.

Description The `__irq` keyword declares an interrupt function. All interrupt functions must be compiled in ARM mode. A function declared `__irq` does not accept parameters and does not have a return value. This keyword is not available when you compile for Cortex-M devices.

Example

```
__irq __arm void interrupt_function(void);
```

`__little_endian`

Syntax Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes, page 303*.

Description The `__little_endian` keyword is used for accessing a variable that is stored in the little-endian byte order regardless of what byte order the rest of the application uses. The `__little_endian` keyword is available when you compile for ARMv6 or higher.

Note that this keyword cannot be used on pointers. Also, this attribute cannot be used on arrays.

Example

```
__little_endian long my_variable;
```

See also *__big_endian, page 307*.

`__nested`

Syntax Follows the generic syntax rules for object attributes that can be used on functions, see *Object attributes, page 305*.

Description The `__nested` keyword modifies the enter and exit code of an interrupt function to allow for nested interrupts. This allows interrupts to be enabled, which means new interrupts can be served inside an interrupt function, without overwriting the `SPSR` and return address in `R14`. Nested interrupts are only supported for `__irq` declared functions.

Note: The `__nested` keyword requires the processor mode to be in either User or System mode.

Example

```
__irq __nested __arm void interrupt_handler(void);
```

See also *Nested interrupts, page 75*.

`__no_init`

Syntax	Follows the generic syntax rules for object attributes, see <i>Object attributes, page 305</i> .
Description	Use the <code>__no_init</code> keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.
Example	<pre>__no_init int myarray[10];</pre>
See also	<i>Do not initialize directive, page 394</i> .

`__noreturn`

Syntax	Follows the generic syntax rules for object attributes, see <i>Object attributes, page 305</i> .
Description	The <code>__noreturn</code> keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are <code>abort</code> and <code>exit</code> .
Example	<pre>__noreturn void terminate(void);</pre>

`__packed`

Syntax	Follows the generic syntax rules for type attributes that can be used on data, see <i>Type attributes, page 303</i> .
Description	<p>Use the <code>__packed</code> keyword to decrease the data type alignment to 1. <code>__packed</code> can be used for two purposes:</p> <ul style="list-style-type: none"> • When used with a <code>struct</code> or <code>union</code> type definition, the maximum alignment of members of that <code>struct</code> or <code>union</code> is set to 1, to eliminate any gaps between the members. The type of each members also receives the <code>__packed</code> type attribute. • When used with any other type, the resulting type is the same as the type without the <code>__packed</code> type attribute, but with an alignment of 1. Types that already have an alignment of 1 are not affected by the <code>__packed</code> type attribute. <p>A normal pointer can be implicitly converted to a pointer to <code>__packed</code>, but the reverse conversion requires a cast.</p> <p>Note: Accessing data types at other alignments than their natural alignment can result in code that is significantly larger and slower.</p>

Example

```
__packed struct X {char ch; int i;};           /* No pad bytes */
void Foo(struct X * xp)                       /* No need for __packed here */
{
    int * p1          = &xp->i; /* Error:"int *">"int __packed *" */
    int __packed * p2 = &xp->i;           /* OK */
    char * p2         = &xp->ch;        /* OK, char not affected */
}
```

See also

pack, page 328.

__ramfunc**Syntax**

Follows the generic syntax rules for object attributes, see *Object attributes*, page 305.

Description

The `__ramfunc` keyword makes a function execute in RAM. Two code sections will be created: one for the RAM execution (`.textrw`), and one for the ROM initialization (`.textrw_init`).

If a function declared `__ramfunc` tries to access ROM, the compiler will issue a warning. This behavior is intended to simplify the creation of *upgrade* routines, for instance, rewriting parts of flash memory. If this is not why you have declared the function `__ramfunc`, you can safely ignore or disable these warnings.

Functions declared `__ramfunc` are by default stored in the section named `.textrw`.

Example

```
__ramfunc int FlashPage(char * data, char * page);
```

See also

To read more about `__ramfunc` declared functions in relation to breakpoints, see the *C-SPY® Debugging Guide for ARM®*.

__root**Syntax**

Follows the generic syntax rules for object attributes, see *Object attributes*, page 305.

Description

A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.


Example

```
__root int myarray[10];
```

See also

For more information about root symbols and how they are kept, see *Keeping symbols and sections*, page 99.

`__stackless`

Syntax	Follows the generic syntax rules for object attributes that can be used on functions, see <i>Object attributes, page 305</i> .
Description	<p>The <code>__stackless</code> keyword declares a function that can be called without a working stack.</p> <div><p>Warning: A function declared <code>__stackless</code> violates the calling convention in such a way that it is not possible to return from it. However, the compiler cannot reliably detect if the function returns and will not issue an error if it does.</p></div>
Example	<pre>__stackless void start_application(void);</pre>

`__swi`

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes, page 303</i> .
Description	<p>The <code>__swi</code> keyword declares a software interrupt function. It inserts an <code>SVC</code> (formerly <code>SWI</code>) instruction and the specified software interrupt number to make a proper function call. A function declared <code>__swi</code> accepts arguments and returns values. The <code>__swi</code> keyword makes the compiler generate the correct return sequence for a specific software interrupt function. Software interrupt functions follow the same calling convention regarding parameters and return values as an ordinary function, except for the stack usage.</p> <p>The <code>__swi</code> keyword also expects a software interrupt number which is specified with the <code>#pragma swi_number=number</code> directive. The <code>swi_number</code> is used as an argument to the generated assembler <code>SVC</code> instruction, and can be used by the <code>SVC</code> interrupt handler, for example <code>SWI_Handler</code>, to select one software interrupt function in a system containing several such functions. Note that the software interrupt number should only be specified in the function declaration—typically, in a header file that you include in the source code file that calls the interrupt function—not in the function definition.</p> <p>Note: All interrupt functions must be compiled in ARM mode, except for Cortex-M. Use either the <code>__arm</code> keyword or the <code>#pragma type_attribute=__arm</code> directive to alter the default behavior if needed.</p>

Example

To declare your software interrupt function, typically in a header file, write for example like this:

```
#pragma swi_number=0x23
__swi int swi0x23_function(int a, int b);
...
```

To call the function:

```
...
int x = swi0x23_function(1, 2); /* Will be replaced by SVC 0x23,
                                hence the linker will never
                                try to locate the
                                swi0x23_function */
...
```

Somewhere in your application source code, you define your software interrupt function:

```
...
__swi __arm int the_actual_swi0x23_function(int a, int b)
{
    ...
    return 42;
}
```

See also

Software interrupts, page 76 and Calling convention, page 153.

__task**Syntax**

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes, page 303*.

Description

This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the start function for a task in an RTOS.

By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared `__task` do not save all registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared `__task`.

Example `__task void my_handler(void);`

__thumb

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes, page 303*.

Description The `__thumb` keyword makes a function execute in Thumb mode. Unless the function is also declared `__interwork`, the function declared `__thumb` can only be called from functions that also execute in Thumb mode.

A function declared `__thumb` cannot be declared `__arm`.

Note: Non-interwork Thumb functions cannot be called from ARM mode.

Example `__thumb int func2(void);`

See also `__interwork, page 308`.

__weak

Syntax Follows the generic syntax rules for object attributes, see *Object attributes, page 305*.

Description Using the `__weak` object attribute on an external declaration of a symbol makes all references to that symbol in the module weak.

Using the `__weak` object attribute on a public definition of a symbol makes that definition a weak definition.

The linker will not include a module from a library solely to satisfy weak references to a symbol, nor will the lack of a definition for a weak reference result in an error. If no definition is included, the address of the object will be zero.

When linking, a symbol can have any number of weak definitions, and at most one non-weak definition. If the symbol is needed, and there is a non-weak definition, this definition will be used. If there is no non-weak definition, one of the weak definitions will be used.

Example

```
extern __weak int foo; /* A weak reference. */

__weak void bar(void) /* A weak definition. */
{
    /* Increment foo if it was included. */
    if (&foo != 0)
        ++foo;}
```

Pragma directives

This chapter describes the pragma directives of the compiler.

The `#pragma` directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The pragma directives control the behavior of the compiler, for example how it allocates memory for variables and functions, whether it allows extended keywords, and whether it outputs warning messages.

The pragma directives are always enabled in the compiler.

Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
<code>bitfields</code>	Controls the order of bitfield members
<code>calls</code>	Lists possible called functions for indirect calls
<code>call_graph_root</code>	Specifies that the function is a call graph root
<code>data_alignment</code>	Gives a variable a higher (more strict) alignment
<code>diag_default</code>	Changes the severity level of diagnostic messages
<code>diag_error</code>	Changes the severity level of diagnostic messages
<code>diag_remark</code>	Changes the severity level of diagnostic messages
<code>diag_suppress</code>	Suppresses diagnostic messages
<code>diag_warning</code>	Changes the severity level of diagnostic messages
<code>error</code>	Signals an error while parsing
<code>include_alias</code>	Specifies an alias for an include file
<code>inline</code>	Controls inlining of a function
<code>language</code>	Controls the IAR Systems language extensions

Table 34: Pragma directives summary

Pragma directive	Description
location	Specifies the absolute address of a variable, places a variable in a register, or places groups of functions or variables in named sections
message	Prints a message
object_attribute	Changes the definition of a variable or a function
optimize	Specifies the type and level of an optimization
pack	Specifies the alignment of structures and union members
__printf_args	Verifies that a function with a printf-style format string is called with the correct arguments
required	Ensures that a symbol that is needed by another symbol is included in the linked output
rtmodel	Adds a runtime model attribute to the module
__scanf_args	Verifies that a function with a scanf-style format string is called with the correct arguments
section	Declares a section name to be used by intrinsic functions
segment	This directive is an alias for #pragma section
STDC CX_LIMITED_RANGE	Specifies whether the compiler can use normal complex mathematical formulas or not
STDC FENV_ACCESS	Specifies whether your source code accesses the floating-point environment or not.
STDC FP_CONTRACT	Specifies whether the compiler is allowed to contract floating-point expressions or not.
swi_number	Sets the interrupt number of a software interrupt function
type_attribute	Changes the declaration and definitions of a variable or function
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable

Table 34: Pragma directives summary (Continued)

Note: For portability reasons, the pragma directives `alignment`, `baseaddr`, `codeseg`, `constseg`, `dataseg`, `function`, `memory`, and `warnings` are recognized but will give a diagnostic message. It is important to be aware of this if you need to port existing code that contains any of those pragma directives. See also *Recognized pragma directives* (6.10.6), page 466.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

bitfields

Syntax	#pragma bitfields=disjoint_types joined_types reversed_disjoint_types reversed default}	
Parameters	disjoint_types	Bitfield members are placed from the least significant bit to the most significant bit in the container type. Storage containers of bitfields with different base types will not overlap.
	joined_types	Bitfield members are placed depending on the byte order. Storage containers of bitfields will overlap other structure members. For more information, see <i>Bitfields</i> , page 304.
	reversed_disjoint_types	Bitfield members are placed from the most significant bit to the least significant bit in the container type. Storage containers of bitfields with different base types will not overlap.
	reversed	This is an alias for <code>reversed_disjoint_types</code> .
	default	Restores to default layout of bitfield members. The default behavior for the compiler is <code>joined_types</code> .
Description	Use this pragma directive to control the layout of bitfield members.	
Example	<pre>#pragma bitfields=disjoint_types /* Structure that uses disjoint bitfield types. */ struct S { unsigned char error : 1; unsigned char size : 4; unsigned short code : 10; }; #pragma bitfields=default /* Restores to default setting. */</pre>	
See also	<i>Bitfields</i> , page 304.	

calls

Syntax	<code>#pragma calls=function[, function...]</code>
Parameters	<i>function</i> Any declared function
Description	Use this pragma directive to list the functions that can be indirectly called in the following statement. This information can be used for stack usage analysis in the linker. Note: For an accurate result, you must list all possible called functions.
Example	<pre>void Fun1(), Fun2(); void Caller(void (*fp)(void)) { #pragma calls = Fun1, Fun2 (*fp)(); }</pre>
See also	<i>Stack usage analysis, page 90</i>

call_graph_root

Syntax	<code>#pragma call_graph_root[=category]</code>
Parameters	<i>category</i> A string that identifies an optional call graph root category
Description	Use this pragma directive to specify that, for stack usage analysis purposes, the immediately following function is a call graph root. You can also specify an optional category. The compiler will usually automatically assign a call graph root category to interrupt and task functions. If you use the <code>#pragma call_graph_root</code> directive on such a function you will override the default category. You can specify any string as a category.
Example	<code>#pragma call_graph_root="interrupt"</code>
See also	<i>Stack usage analysis, page 90</i>

data_alignment

Syntax	<code>#pragma data_alignment=<i>expression</i></code>	
Parameters	<i>expression</i>	A constant which must be a power of two (1, 2, 4, etc.).
Description	<p>Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.</p> <p>When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.</p> <p>Note: Normally, the size of a variable is a multiple of its alignment. The <code>data_alignment</code> directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.</p>	

diag_default

Syntax	<code>#pragma diag_default=<i>tag</i>[, <i>tag</i>, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number <code>Pe177</code> .
Description	<p>Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options <code>--diag_error</code>, <code>--diag_remark</code>, <code>--diag_suppress</code>, or <code>--diag_warnings</code>, for the diagnostic messages specified with the tags.</p>	
See also	<i>Diagnostics, page 228.</i>	

diag_error

Syntax	<code>#pragma diag_error=<i>tag</i>[, <i>tag</i>, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number <code>Pe177</code> .

Description	Use this pragma directive to change the severity level to <code>error</code> for the specified diagnostics.
See also	<i>Diagnostics, page 228.</i>

diag_remark

Syntax	#pragma diag_remark=tag[, tag, ...]	
Parameters	tag	The number of a diagnostic message, for example the message number Pe177.
Description	Use this pragma directive to change the severity level to <code>remark</code> for the specified diagnostic messages.	
See also	<i>Diagnostics, page 228.</i>	

diag_suppress

Syntax	#pragma diag_suppress=tag[, tag, ...]	
Parameters	tag	The number of a diagnostic message, for example the message number Pe117.
Description	Use this pragma directive to suppress the specified diagnostic messages.	
See also	<i>Diagnostics, page 228.</i>	

diag_warning

Syntax	#pragma diag_warning=tag[, tag, ...]	
Parameters	tag	The number of a diagnostic message, for example the message number Pe826.
Description	Use this pragma directive to change the severity level to <code>warning</code> for the specified diagnostic messages.	

See also *Diagnostics, page 228.*

error

Syntax `#pragma error message`

Parameters

message A string that represents the error message.

Description

Use this pragma directive to cause an error message when it is parsed. This mechanism is different from the preprocessor directive `#error`, because the `#pragma error` directive can be included in a preprocessor macro using the `_Pragma` form of the directive and only causes an error if the macro is used.

Example

```
#if FOO_AVAILABLE
#define FOO ...
#else
#define FOO _Pragma("error\"Foo is not available\"")
#endif
```

If `FOO_AVAILABLE` is zero, an error will be signaled if the `FOO` macro is used in actual source code.

include_alias

Syntax `#pragma include_alias ("orig_header" , "subst_header")`
`#pragma include_alias (<orig_header> , <subst_header>)`

Parameters

orig_header The name of a header file for which you want to create an alias.
subst_header The alias for the original header file.

Description

Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

See also *Include file search procedure, page 225.*

inline

Syntax `#pragma inline[=forced|=never]`

Parameters	No parameter	Has the same effect as the <code>inline</code> keyword.
	<code>forced</code>	Disables the compiler's heuristics and forces inlining.
	<code>never</code>	Disables the compiler's heuristics and makes sure that the function will not be inlined.

Description Use `#pragma inline` to advise the compiler that the function whose declaration follows immediately after the directive should be expanded inline into the body of the calling function. Whether the inlining actually occurs is subject to the compiler's heuristics.

`#pragma inline` is similar to the C++ keyword `inline`. The difference is that the compiler uses C++ `inline` semantics for the `#pragma inline` directive, but uses the Standard C semantics for the `inline` keyword.

Specifying `#pragma inline=never` disables the compiler's heuristics and makes sure that the function will not be inlined.

Specifying `#pragma inline=forced` will force the compiler to treat the function declared immediately after the directive as a candidate for inlining, regardless of the compiler's heuristics. If the compiler fails to inline the candidate function for some reason, for example due to recursion, a warning message is emitted.

Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` will enable inlining of the function in question also on the Medium optimization level.

See also *Function inlining, page 212.*

language

Syntax	<code>#pragma language={extended default save restore}</code>	
Parameters	extended	Enables the IAR Systems language extensions from the first use of the pragma directive and onward.
	default	From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.
	save restore	Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code. Each use of <code>save</code> must be followed by a matching <code>restore</code> in the same file without any intervening <code>#include</code> directive.
Description	Use this pragma directive to control the use of language extensions.	
Example 1	At the top of a file that needs to be compiled with IAR Systems extensions enabled: <pre>#pragma language=extended /* The rest of the file. */</pre>	
Example 2	Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled, but where the state before the sequence cannot be assumed to be the same as that specified by the compiler options in use: <pre>#pragma language=save #pragma language=extended /* Part of source code. */ #pragma language=restore</pre>	
See also	<i>-e</i> , page 249 and <i>--strict</i> , page 270.	

location

Syntax	<code>#pragma location={address register NAME}</code>	
Parameters	address	The absolute address of the global or static variable or function for which you want an absolute location.
	register	An identifier that corresponds to one of the ARM core registers R4–R11.

	<i>NAME</i>	A user-defined section name; cannot be a section name predefined for use by the compiler and linker.
Description	<p>Use this pragma directive to specify:</p> <ul style="list-style-type: none">• The location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variables must be declared <code>__no_init</code>.• An identifier specifying a register. The variable defined after the pragma directive is placed in the register. The variable must be declared as <code>__no_init</code> and have file scope. <p>A string specifying a section for placing either a variable or function whose declaration follows the pragma directive. Do not place variables that would normally be in different sections (for example, variables declared as <code>__no_init</code> and variables declared as <code>const</code>) in the same named section.</p>	
Example	<pre>#pragma location=0xFFFF0400 __no_init volatile char PORT1; /* PORT1 is located at address 0xFFFF0400 */ #pragma location=R8 __no_init int TASK; /* TASK is placed in R8 */ #pragma section="FLASH" #pragma location="FLASH" char PORT2; /* PORT2 is located in section FLASH */ /* A better way is to use a corresponding mechanism */ #define FLASH _Pragma("location=\"FLASH\"") FLASH int i; /* i is placed in the FLASH section */</pre>	
See also	<i>Controlling data and function placement in memory, page 205.</i>	
message		
Syntax	<code>#pragma message(<i>message</i>)</code>	
Parameters	<i>message</i>	The message that you want to direct to the standard output stream.

Description Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax `#pragma object_attribute=object_attribute[,object_attribute,...]`

Parameters For information about object attributes that can be used with this pragma directive, see *Object attributes*, page 305.

Description Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.

Example

```
#pragma object_attribute=__no_init
char bar;
```

See also *General syntax rules for extended keywords*, page 303.

optimize

Syntax `#pragma optimize=[goal][level][no_optimization...]`

Parameters

<i>goal</i>	Choose between: balanced, optimizes balanced between speed and size size, optimizes for size speed, optimizes for speed.
<i>level</i>	Specifies the level of optimization; choose between none, low, medium, or high.

	<div><div><code>no_optimization</code></div><div>Disables one or several optimizations; choose between: <code>no_code_motion</code>, disables code motion <code>no_cse</code>, disables common subexpression elimination <code>no_inline</code>, disables function inlining <code>no_tbaa</code>, disables type-based alias analysis <code>no_unroll</code>, disables loop unrolling <code>no_scheduling</code>, disables instruction scheduling.</div></div>						
Description	<div><p>Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.</p><p>The parameters <code>speed</code>, <code>size</code>, and <code>balanced</code> only have effect on the <code>high</code> optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.</p><p>Note: If you use the <code>#pragma optimize</code> directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.</p></div>						
Example	<div><pre>#pragma optimize=speed int SmallAndUsedOften() { /* Do something here. */ } #pragma optimize=size int BigAndSeldomUsed() { /* Do something here. */ }</pre></div>						
pack							
Syntax	<div><pre>#pragma pack(n) #pragma pack() #pragma pack({push pop}[, name] [, n])</pre></div>						
Parameters	<div><table><tr><td><code>n</code></td><td>Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16</td></tr><tr><td>Empty list</td><td>Restores the structure alignment to default</td></tr><tr><td><code>push</code></td><td>Sets a temporary structure alignment</td></tr></table></div>	<code>n</code>	Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16	Empty list	Restores the structure alignment to default	<code>push</code>	Sets a temporary structure alignment
<code>n</code>	Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16						
Empty list	Restores the structure alignment to default						
<code>push</code>	Sets a temporary structure alignment						

	<code>pop</code>	Restores the structure alignment from a temporarily pushed alignment
	<code>name</code>	An optional pushed or popped alignment label
Description	<p>Use this pragma directive to specify the maximum alignment of <code>struct</code> and <code>union</code> members.</p> <p>The <code>#pragma pack</code> directive affects declarations of structures following the pragma directive to the next <code>#pragma pack</code> or the end of the compilation unit.</p> <p>Note: This can result in significantly larger and slower code when accessing members of the structure.</p>	
See also	<i>Structure types, page 311</i> and <i>__packed, page 310</i> .	

__printf_args

Syntax	<code>#pragma __printf_args</code>
Description	Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example <code>%d</code>) is syntactically correct.
Example	<pre>#pragma __printf_args int printf(char const *,...); void PrintNumbers(unsigned short x) { printf("%d", x); /* Compiler checks that x is an integer */ }</pre>

required

Syntax	<code>#pragma required=symbol</code>
Parameters	<p><i>symbol</i></p> <p>Any statically linked function or variable.</p>
Description	Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the section it resides in.

Example

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
    /* Do something here. */
}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax

```
#pragma rtmodel="key", "value"
```

Parameters

"key"	A text string that specifies the runtime model attribute.
"value"	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value *. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C", "ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 141.

__scanf_args

Syntax	<code>#pragma __scanf_args</code>
Description	Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct.
Example	<pre>#pragma __scanf_args int scanf(char const *,...); int GetNumber() { int nr; scanf("%d", &nr); /* Compiler checks that the argument is a pointer to an integer */ return nr; }</pre>

section

Syntax	<code>#pragma section="NAME" [align]</code> <code>alias</code> <code>#pragma segment="NAME" [align]</code>				
Parameters	<table><tr><td><i>NAME</i></td><td>The name of the section.</td></tr><tr><td><i>align</i></td><td>Specifies an alignment for the section part. The value must be a constant integer expression to the power of two.</td></tr></table>	<i>NAME</i>	The name of the section.	<i>align</i>	Specifies an alignment for the section part. The value must be a constant integer expression to the power of two.
<i>NAME</i>	The name of the section.				
<i>align</i>	Specifies an alignment for the section part. The value must be a constant integer expression to the power of two.				
Description	Use this pragma directive to define a section name that can be used by the section operators __section_begin, __section_end, and __section_size. All section declarations for a specific section must have the same alignment.				
Example	<code>#pragma section="MYSECTION" 4</code>				
See also	<i>Dedicated section operators, page 169.</i> For more information about sections and segment parts, see the chapter <i>Linking your application</i> .				

STDC CX_LIMITED_RANGE

Syntax	#pragma STDC CX_LIMITED_RANGE {ON OFF DEFAULT}	
Parameters	ON	Normal complex mathematic formulas can be used.
	OFF	Normal complex mathematic formulas cannot be used.
	DEFAULT	Sets the default behavior, that is OFF.
Description	Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for * (multiplication), / (division), and abs. Note: This directive is required by Standard C. The directive is recognized but has no effect in the compiler.	

STDC FENV_ACCESS

Syntax	#pragma STDC FENV_ACCESS {ON OFF DEFAULT}	
Parameters	ON	Source code accesses the floating-point environment. Note that this argument is not supported by the compiler.
	OFF	Source code does not access the floating-point environment.
	DEFAULT	Sets the default behavior, that is OFF.
Description	Use this pragma directive to specify whether your source code accesses the floating-point environment or not. Note: This directive is required by Standard C.	

STDC FP_CONTRACT

Syntax	#pragma STDC FP_CONTRACT {ON OFF DEFAULT}	
Parameters	ON	The compiler is allowed to contract floating-point expressions.
	OFF	The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler.
	DEFAULT	Sets the default behavior, that is ON.

Description Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C.

Example `#pragma STDC FP_CONTRACT=ON`

swi_number

Syntax `#pragma swi_number=number`

Parameters

<i>number</i>	The software interrupt number
---------------	-------------------------------

Description Use this pragma directive together with the `__swi` extended keyword. It is used as an argument to the generated `SWC` assembler instruction, and is used for selecting one software interrupt function in a system containing several such functions.

Example `#pragma swi_number=17`

See also *Software interrupts, page 76.*

type_attribute

Syntax `#pragma type_attribute=type_attribute[, type_attribute, ...]`

Parameters For information about type attributes that can be used with this pragma directive, see *Type attributes, page 303.*

Description Use this pragma directive to specify IAR-specific *type attributes*, which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example In this example, thumb-mode code is generated for the function `foo`:

```
#pragma type_attribute=__thumb
void foo(void)
{
}
```

This declaration, which uses extended keywords, is equivalent:

```
__thumb void foo(void)
{
}
```

See also The chapter *Extended keywords* for more information.

weak

Syntax	#pragma weak <i>symbol1</i> ={ <i>symbol2</i> }	
Parameters	<i>symbol1</i>	A function or variable with external linkage.
	<i>symbol2</i>	A defined function or variable.
Description	<p>This pragma directive can be used in one of two ways:</p> <ul style="list-style-type: none">● To make the definition of a function or variable with external linkage a weak definition. The <code>__weak</code> attribute can also be used for this purpose.● To create a weak alias for another function or variable. You can make more than one alias for the same function or variable.	
Example	<p>To make the definition of <code>foo</code> a weak definition, write:</p> <pre>#pragma weak foo</pre> <p>To make <code>NMI_Handler</code> a weak alias for <code>Default_Handler</code>, write:</p> <pre>#pragma weak NMI_Handler=Default_Handler</pre> <p>If <code>NMI_Handler</code> is not defined elsewhere in the program, all references to <code>NMI_Handler</code> will refer to <code>Default_Handler</code>.</p>	
See also	<code>__weak</code> , page 314.	

Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

To use Neon intrinsic functions in an application, include the header file `arm_neon.h`. For more information, see *Intrinsic functions for Neon instructions*, page 345.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

This table summarizes the intrinsic functions:

Intrinsic function	Description
<code>__CLREX</code>	Inserts a CLREX instruction
<code>__CLZ</code>	Inserts a CLZ instruction
<code>__disable_fiq</code>	Disables fast interrupt requests (fiq)
<code>__disable_interrupt</code>	Disables interrupts
<code>__disable_irq</code>	Disables interrupt requests (irq)
<code>__DMB</code>	Inserts a DMB instruction
<code>__DSB</code>	Inserts a DSB instruction
<code>__enable_fiq</code>	Enables fast interrupt requests (fiq)
<code>__enable_interrupt</code>	Enables interrupts
<code>__enable_irq</code>	Enables interrupt requests (irq)
<code>__get_BASEPRI</code>	Returns the value of the Cortex-M3/Cortex-M4 BASEPRI register
<code>__get_CONTROL</code>	Returns the value of the Cortex-M CONTROL register

Table 35: Intrinsic functions summary

Intrinsic function	Description
__get_CPSR	Returns the value of the ARM CPSR (Current Program Status Register)
__get_FAULTMASK	Returns the value of the Cortex-M3/Cortex-M4 FAULTMASK register
__get_FPSCR	Returns the value of FPSCR
__get_interrupt_state	Returns the interrupt state
__get_IPSR	Returns the value of the IPSR register
__get_LR	Returns the value of the link register
__get_MSP	Returns the value of the MSP register
__get_PRIMASK	Returns the value of the Cortex-M PRIMASK register
__get_PSP	Returns the value of the PSP register
__get_PSR	Returns the value of the PSR register
__get_SB	Returns the value of the static base register
__get_SP	Returns the value of the stack pointer register
__ISB	Inserts an ISB instruction
__LDC, __LDCL, __LDC2, __LDC2L	Inserts the coprocessor load instruction LDC
__LDC, __LDCL, __LDC2, __LDC2L	Inserts the coprocessor load instruction LDCL
__LDC, __LDCL, __LDC2, __LDC2L	Inserts the coprocessor load instruction LDC2
__LDC, __LDCL, __LDC2, __LDC2L	Inserts the coprocessor load instruction LDC2L
__LDC_noidx, __LDCL_noidx, __LDC2_noidx, __LDC2L_noidx	Inserts the coprocessor load instruction LDC
__LDC_noidx, __LDCL_noidx, __LDC2_noidx, __LDC2L_noidx	Inserts the coprocessor load instruction LDCL
__LDC_noidx, __LDCL_noidx, __LDC2_noidx, __LDC2L_noidx	Inserts the coprocessor load instruction LDC2
__LDC_noidx, __LDCL_noidx, __LDC2_noidx, __LDC2L_noidx	Inserts the coprocessor load instruction LDC2L
__LDREX, __LDREXB, __LDREXD, __LDREXH	Inserts an LDREX instruction
__LDREX, __LDREXB, __LDREXD, __LDREXH	Inserts an LDREXB instruction

Table 35: Intrinsic functions summary (Continued)

Intrinsic function	Description
<code>__LDREX, __LDREXB, __LDREXD, __LDREXH</code>	Inserts an LDREXD instruction
<code>__LDREX, __LDREXB, __LDREXD, __LDREXH</code>	Inserts an LDREXH instruction
<code>__MCR</code>	Inserts the coprocessor write instruction MCR
<code>__MRC</code>	Inserts the coprocessor read instruction MRC
<code>__no_operation</code>	Inserts a NOP instruction
<code>__PKHBT</code>	Inserts a PKHBT instruction
<code>__PKHTB</code>	Inserts a PKHTB instruction
<code>__QADD, __QDADD, __QDSUB, __QSUB</code>	Inserts a QADD instruction
<code>__QADD8, __QADD16, __QASX, __QSAX, __QSUB8, __QSUB16</code>	Inserts a QADD8 instruction
<code>__QADD8, __QADD16, __QASX, __QSAX, __QSUB8, __QSUB16</code>	Inserts a QADD16 instruction
<code>__QADD8, __QADD16, __QASX, __QSAX, __QSUB8, __QSUB16</code>	Inserts a QASX instruction
<code>__QCFflag</code>	Returns the value of the cumulative saturation flag of the FPSCR register
<code>__QADD, __QDADD, __QDSUB, __QSUB</code>	Inserts a QDADD instruction
<code>__QDOUBLE</code>	Inserts a QADD instruction
<code>__QADD, __QDADD, __QDSUB, __QSUB</code>	Inserts a QDSUB instruction
<code>__QFlag</code>	Returns the Q flag that indicates if overflow/saturation has occurred
<code>__QADD8, __QADD16, __QASX, __QSAX, __QSUB8, __QSUB16</code>	Inserts a QSAX instruction
<code>__QADD, __QDADD, __QDSUB, __QSUB</code>	Inserts a QSUB instruction
<code>__QADD8, __QADD16, __QASX, __QSAX, __QSUB8, __QSUB16</code>	Inserts a QSUB8 instruction
<code>__QADD8, __QADD16, __QASX, __QSAX, __QSUB8, __QSUB16</code>	Inserts a QSUB16 instruction
<code>__RBIT</code>	Inserts an RBIT instruction
<code>__reset_Q_flag</code>	Clears the Q flag that indicates if overflow/saturation has occurred

Table 35: Intrinsic functions summary (Continued)

Intrinsic function	Description
<code>__reset_QC_flag</code>	Clears the value of the cumulative saturation flag QC of the FPSCR register
<code>__REV, __REV16, __REVSH</code>	Inserts an REV instruction
<code>__REV, __REV16, __REVSH</code>	Inserts an REV16 instruction
<code>__REV, __REV16, __REVSH</code>	Inserts an REVSH instruction
<code>__SADD8, __SADD16, __SASX, __SSAX, __SSUB8, __SSUB16</code>	Inserts an SADD8 instruction
<code>__SADD8, __SADD16, __SASX, __SSAX, __SSUB8, __SSUB16</code>	Inserts an SADD16 instruction
<code>__SADD8, __SADD16, __SASX, __SSAX, __SSUB8, __SSUB16</code>	Inserts an SASX instruction
<code>__SEL</code>	Inserts an SEL instruction
<code>__set_BASEPRI</code>	Sets the value of the Cortex-M3/Cortex-M4 BASEPRI register
<code>__set_CONTROL</code>	Sets the value of the Cortex-M CONTROL register
<code>__set_CPSR</code>	Sets the value of the ARM CPSR (Current Program Status Register)
<code>__set_FAULTMASK</code>	Sets the value of the Cortex-M3/Cortex-M4 FAULTMASK register
<code>__set_FPSCR</code>	Sets the value of the FPSCR register
<code>__set_interrupt_state</code>	Restores the interrupt state
<code>__set_LR</code>	Assigns a new address to the link register
<code>__set_MSP</code>	Sets the value of the MSP register
<code>__set_PRIMASK</code>	Sets the value of the Cortex-M PRIMASK register
<code>__set_PSP</code>	Sets the value of the PSP register
<code>__set_SB</code>	Assigns a new address to the static base register
<code>__set_SP</code>	Assigns a new address to the stack pointer register
<code>__SEV</code>	Inserts an SEV instruction
<code>__SHADD8, __SHADD16, __SHASX, __SHSAX, __SHSUB8, __SHSUB16</code>	Inserts an SHADD8 instruction

Table 35: Intrinsic functions summary (Continued)

Intrinsic function	Description
__SHADD8, __SHADD16, __SHASX, __S HSAX, __SHSUB8, __SHSUB16	Inserts an SHADD16 instruction
__SHADD8, __SHADD16, __SHASX, __S HSAX, __SHSUB8, __SHSUB16	Inserts an SHASX instruction
__SHADD8, __SHADD16, __SHASX, __S HSAX, __SHSUB8, __SHSUB16	Inserts an SHSAX instruction
__SHADD8, __SHADD16, __SHASX, __S HSAX, __SHSUB8, __SHSUB16	Inserts an SHSUB8 instruction
__SHADD8, __SHADD16, __SHASX, __S HSAX, __SHSUB8, __SHSUB16	Inserts an SHSUB16 instruction
__SMLABB, __SMLABT, __SMLATB, __S MLATT, __SMLAWB, __SMLAWT	Inserts an SMLABB instruction
__SMLABB, __SMLABT, __SMLATB, __S MLATT, __SMLAWB, __SMLAWT	Inserts an SMLABT instruction
__SMLAD, __SMLADX, __SMLSD, __SML SDX	Inserts an SMLAD instruction
__SMLAD, __SMLADX, __SMLSD, __SML SDX	Inserts an SMLADX instruction
__SMLALBB, __SMLALBT, __SMLALTB, __S MLALTT	Inserts an SMLALBB instruction
__SMLALBB, __SMLALBT, __SMLALTB, __S MLALTT	Inserts an SMLALBT instruction
__SMLALD, __SMLALDX, __SMLS LD, __SML SMLS LD	Inserts an SMLALD instruction
__SMLALD, __SMLALDX, __SMLS LD, __SML SMLS LD	Inserts an SMLALDX instruction
__SMLALBB, __SMLALBT, __SMLALTB, __S MLALTT	Inserts an SMLALTB instruction
__SMLALBB, __SMLALBT, __SMLALTB, __S MLALTT	Inserts an SMLALTT instruction
__SMLABB, __SMLABT, __SMLATB, __S MLATT, __SMLAWB, __SMLAWT	Inserts an SMLATB instruction
__SMLABB, __SMLABT, __SMLATB, __S MLATT, __SMLAWB, __SMLAWT	Inserts an SMLATT instruction
__SMLABB, __SMLABT, __SMLATB, __S MLATT, __SMLAWB, __SMLAWT	Inserts an SMLAWB instruction

Table 35: Intrinsic functions summary (Continued)

Intrinsic function	Description
__SMLABB, __SMLABT, __SMLATB, __S MLATT, __SMLAWB, __SMLAWT	Inserts an SMLAWT instruction
__SMLAD, __SMLADX, __SMLSD, __SML SDX	Inserts an SMLSD instruction
__SMLAD, __SMLADX, __SMLSD, __SML SDX	Inserts an SMLSXD instruction
__SMLALD, __SMLALDX, __SMLS LD, __ SMLS LDX	Inserts an SMLS LD instruction
__SMLALD, __SMLALDX, __SMLS LD, __ SMLS LDX	Inserts an SMLS LDX instruction
__SMMLA, __SMMLAR, __SMMLS, __SMM LSR	Inserts an SMMLA instruction
__SMMLA, __SMMLAR, __SMMLS, __SMM LSR	Inserts an SMMLAR instruction
__SMMLA, __SMMLAR, __SMMLS, __SMM LSR	Inserts an SMMLS instruction
__SMMLA, __SMMLAR, __SMMLS, __SMM LSR	Inserts an SMMLS instruction
__SMMLA, __SMMLAR, __SMMLS, __SMM LSR	Inserts an SMMLSR instruction
__SMMUL, __SMMULR	Inserts an SMMUL instruction
__SMMUL, __SMMULR	Inserts an SMMULR instruction
__SMUAD, __SMUADX, __SMUSD, __SMU SDX	Inserts an SMUAD instruction
__SMUAD, __SMUADX, __SMUSD, __SMU SDX	Inserts an SMUADX instruction
__SMUL	Inserts a signed 16-bit multiplication
__SMULBB, __SMULBT, __SMULTB, __S MULTT, __SMULWB, __SMULWT	Inserts an SMULBB instruction
__SMULBB, __SMULBT, __SMULTB, __S MULTT, __SMULWB, __SMULWT	Inserts an SMULBT instruction
__SMULBB, __SMULBT, __SMULTB, __S MULTT, __SMULWB, __SMULWT	Inserts an SMULTB instruction
__SMULBB, __SMULBT, __SMULTB, __S MULTT, __SMULWB, __SMULWT	Inserts an SMULTT instruction
__SMULBB, __SMULBT, __SMULTB, __S MULTT, __SMULWB, __SMULWT	Inserts an SMULWB instruction

Table 35: Intrinsic functions summary (Continued)

Intrinsic function	Description
__SMULBB, __SMULBT, __SMULTB, __SMULTT, __SMULWB, __SMULWT	Inserts an SMULWT instruction
__SMUAD, __SMUADX, __SMUSD, __SMUSD, __SMUSD	Inserts an SMUSD instruction
__SMUAD, __SMUADX, __SMUSD, __SMUSD, __SMUSD	Inserts an SMUSD instruction
__SSAT	Inserts an SSAT instruction
__SSAT16	Inserts an SSAT16 instruction
__SADD8, __SADD16, __SASX, __SSAX, __SSUB8, __SSUB16	Inserts an SSAX instruction
__SADD8, __SADD16, __SASX, __SSAX, __SSUB8, __SSUB16	Inserts an SSUB8 instruction
__SADD8, __SADD16, __SASX, __SSAX, __SSUB8, __SSUB16	Inserts an SSUB16 instruction
__STC, __STCL, __STC2, __STC2L	Inserts the coprocessor store instruction STC
__STC, __STCL, __STC2, __STC2L	Inserts the coprocessor store instruction STCL
__STC, __STCL, __STC2, __STC2L	Inserts the coprocessor store instruction STC2
__STC, __STCL, __STC2, __STC2L	Inserts the coprocessor store instruction STC2L
__STC_nidx, __STCL_nidx, __STC2_nidx, __STC2L_nidx	Inserts the coprocessor store instruction STC
__STC_nidx, __STCL_nidx, __STC2_nidx, __STC2L_nidx	Inserts the coprocessor store instruction STCL
__STC_nidx, __STCL_nidx, __STC2_nidx, __STC2L_nidx	Inserts the coprocessor store instruction STC2
__STC_nidx, __STCL_nidx, __STC2_nidx, __STC2L_nidx	Inserts the coprocessor store instruction STC2L
__STREX, __STREXB, __STREXD, __STREXH	Inserts a STREX instruction
__STREX, __STREXB, __STREXD, __STREXH	Inserts a STREXB instruction
__STREX, __STREXB, __STREXD, __STREXH	Inserts a STREXD instruction
__STREX, __STREXB, __STREXD, __STREXH	Inserts a STREXH instruction

Table 35: Intrinsic functions summary (Continued)

Intrinsic function	Description
__SWP, __SWPB	Inserts an SWP instruction
__SWP, __SWPB	Inserts an SWPB instruction
__SXTAB, __SXTAB16, __SXTAH, __SXTB16	Inserts an SXTAB instruction
__SXTAB, __SXTAB16, __SXTAH, __SXTB16	Inserts an SXTAB16 instruction
__SXTAB, __SXTAB16, __SXTAH, __SXTB16	Inserts an SXTAH instruction
__SXTAB, __SXTAB16, __SXTAH, __SXTB16	Inserts an SXTB16 instruction
__UADD8, __UADD16, __UASX, __USAX, __USUB8, __USUB16	Inserts a UADD8 instruction
__UADD8, __UADD16, __UASX, __USAX, __USUB8, __USUB16	Inserts a UADD16 instruction
__UADD8, __UADD16, __UASX, __USAX, __USUB8, __USUB16	Inserts a UASX instruction
__UHADD8, __UHADD16, __UHASX, __UHSAX, __UHSUB8, __UHSUB16	Inserts a UHADD8 instruction
__UHADD8, __UHADD16, __UHASX, __UHSAX, __UHSUB8, __UHSUB16	Inserts a UHADD16 instruction
__UHADD8, __UHADD16, __UHASX, __UHSAX, __UHSUB8, __UHSUB16	Inserts a UHASX instruction
__UHADD8, __UHADD16, __UHASX, __UHSAX, __UHSUB8, __UHSUB16	Inserts a UHSAX instruction
__UHADD8, __UHADD16, __UHASX, __UHSAX, __UHSUB8, __UHSUB16	Inserts a UHSUB8 instruction
__UHADD8, __UHADD16, __UHASX, __UHSAX, __UHSUB8, __UHSUB16	Inserts a UHSUB16 instruction
__UMAAL	Inserts a UMAAL instruction
__UQADD8, __UQADD16, __UQASX, __UQSAX, __UQSUB8, __UQSUB16	Inserts a UQADD8 instruction
__UQADD8, __UQADD16, __UQASX, __UQSAX, __UQSUB8, __UQSUB16	Inserts a UQADD16 instruction
__UQADD8, __UQADD16, __UQASX, __UQSAX, __UQSUB8, __UQSUB16	Inserts a UQASX instruction

Table 35: Intrinsic functions summary (Continued)

Intrinsic function	Description
__UQADD8, __UQADD16, __UQASX, __UQSAX, __UQSUB8, __UQSUB16	Inserts a UQSAX instruction
__UQADD8, __UQADD16, __UQASX, __UQSAX, __UQSUB8, __UQSUB16	Inserts a UQSUB8 instruction
__UQADD8, __UQADD16, __UQASX, __UQSAX, __UQSUB8, __UQSUB16	Inserts a UQSUB16 instruction
__USAD8, __USADA8	Inserts a USAD8 instruction
__USAD8, __USADA8	Inserts a USAD8A8 instruction
__USAT	Inserts a USAT instruction
__USAT16	Inserts a USAT16 instruction
__UADD8, __UADD16, __UASX, __USAX, __USUB8, __USUB16	Inserts a USAX instruction
__UADD8, __UADD16, __UASX, __USAX, __USUB8, __USUB16	Inserts a USUB8 instruction
__UADD8, __UADD16, __UASX, __USAX, __USUB8, __USUB16	Inserts a USUB16 instruction
__UXTAB, __UXTAB16, __UXTAH, __UXTB16	Inserts a UXTAB instruction
__UXTAB, __UXTAB16, __UXTAH, __UXTB16	Inserts a UXTAB16 instruction
__UXTAB, __UXTAB16, __UXTAH, __UXTB16	Inserts a UXTAH instruction
__UXTAB, __UXTAB16, __UXTAH, __UXTB16	Inserts a UXTB16 instruction
__WFE	Inserts a WFE instruction
__WFI	Inserts a WFI instruction
__YIELD	Inserts a YIELD instruction

Table 35: Intrinsic functions summary (Continued)

INTRINSIC FUNCTIONS FOR NEON INSTRUCTIONS

The Neon co-processor implements the Advanced SIMD instruction set extension, as defined by the ARM architecture. To use Neon intrinsic functions in an application, include the header file `arm_neon.h`. The functions use vector types that are named according to this pattern:

`<type><size>x<number_of_lanes>_t`

where:

- *type* is int, unsigned int, float, or poly
- *size* is 8, 16, 32, or 64
- *number_of_lanes* is 1, 2, 4, 8, or 16.

The total bit width of a vector type is *size* times *number_of_lanes*, and should fit in a D register (64 bits) or a Q register (128 bits).

For example:

```
__intrinsic float32x2_t vsub_f32(float32x2_t, float32x2_t);
```

The intrinsic function `vsub_f32` inserts a `VSUB.F32` instruction that operates on two 64-bit vectors (D registers), each with two elements (lanes) of 32-bit floating-point type.

Some functions use an array of vector types. As an example, the definition of an array type with four elements of type `float32x2_t` is:

```
typedef struct
{
    float32x2_t val[4];
}
float32x2x4_t;
```

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

__CLREX

Syntax

```
void __CLREX(void);
```

Description

Inserts a `CLREX` instruction.

This intrinsic function requires architecture ARMv6K or ARMv7 for ARM mode, and AVRv7 for Thumb mode.

__CLZ

Syntax

```
unsigned char __CLZ(unsigned long);
```

Description

Inserts a `CLZ` instruction.

This intrinsic function requires an ARMv5 architecture or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.

__disable_fiq

Syntax

```
void __disable_fiq(void);
```

Description

Disables fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode and is not available for Cortex-M devices.

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts. For Cortex-M devices, it raises the execution priority level to 0 by setting the priority mask bit, `PRIMASK`. For other devices, it disables interrupt requests (irq) and fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode.

__disable_irq

Syntax

```
void __disable_irq(void);
```

Description

Disables interrupt requests (irq).

This intrinsic function can only be used in privileged mode and is not available for Cortex-M devices.

__DMB

Syntax

```
void __DMB(void);
```

Description

Inserts a `DMB` instruction. This intrinsic function requires an ARMv6M architecture, or an ARMv7 architecture or higher.

__DSB

Syntax

```
void __DSB(void);
```

Description

Inserts a `DSB` instruction. This intrinsic function requires an ARMv6M architecture, or an ARMv7 architecture or higher.

__enable_fiq

Syntax

```
void __enable_fiq(void);
```

Description

Enables fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts. For Cortex-M devices, it resets the execution priority level to default by clearing the priority mask bit, `PRIMASK`. For other devices, it enables interrupt requests (irq) and fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode.

__enable_irq

Syntax

```
void __enable_irq(void);
```

Description

Enables interrupt requests (irq).

This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

__get_BASEPRI

Syntax

```
unsigned long __get_BASEPRI(void);
```

Description

Returns the value of the `BASEPRI` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 or Cortex-M4 device.

__get_CONTROL

Syntax	<code>unsigned long __get_CONTROL(void);</code>
Description	Returns the value of the <code>CONTROL</code> register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__get_CPSR

Syntax	<code>unsigned long __get_CPSR(void);</code>
Description	Returns the value of the ARM <code>CPSR</code> (Current Program Status Register). This intrinsic function can only be used in privileged mode, is not available for Cortex-M devices, and it requires ARM mode.

__get_FAULTMASK

Syntax	<code>unsigned long __get_FAULTMASK(void);</code>
Description	Returns the value of the <code>FAULTMASK</code> register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 or Cortex-M4 device.

__get_FPSCR

Syntax	<code>unsigned long __get_FPSCR(void);</code>
Description	Returns the value of <code>FPSCR</code> (floating-point status and control register). This intrinsic function is only available for devices with a VFP coprocessor.

__get_interrupt_state

Syntax	<code>__istate_t __get_interrupt_state(void);</code>
Description	Returns the global interrupt state. The return value can be used as an argument to the <code>__set_interrupt_state</code> intrinsic function, which will restore the interrupt state. This intrinsic function can only be used in privileged mode, and cannot be used when using the <code>--aeabi</code> compiler option.

Example

```
#include "intrinsics.h"

void CriticalFn()
{
    __istate_t s = __get_interrupt_state();
    __disable_interrupt();

    /* Do something here. */

    __set_interrupt_state(s);
}
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

__get_IPSR

Syntax

```
unsigned long __get_IPSR(void);
```

Description

Returns the value of the `IPSR` register (Interrupt Program Status Register). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

__get_LR

Syntax

```
unsigned long __get_LR(void);
```

Description

Returns the value of the link register (R14).

__get_MSP

Syntax

```
unsigned long __get_MSP(void);
```

Description

Returns the value of the `MSP` register (Main Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

__get_PRIMASK

Syntax `unsigned long __get_PRIMASK(void);`

Description Returns the value of the `PRIMASK` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__get_PSP

Syntax `unsigned long __get_PSP(void);`

Description Returns the value of the `PSP` register (Process Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

__get_PSR

Syntax `unsigned long __get_PSR(void);`

Description Returns the value of the `PSR` register (combined Program Status Register). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

__get_SB

Syntax `unsigned long __get_SB(void);`

Description Returns the value of the static base register (`R9`).

__get_SP

Syntax `unsigned long __get_SP(void);`

Description Returns the value of the stack pointer register (`R13`).

__ISB

Syntax	<code>void __ISB(void);</code>
Description	Inserts an ISB instruction. This intrinsic function requires an ARMv6M architecture, or an ARMv7 architecture or higher.

**__LDC
__LDCL
__LDC2
__LDC2L**

Syntax	<code>void __LDCxxx(__ul coproc, __ul CRn, __ul const *src);</code>						
Parameters	<table><tr><td><i>coproc</i></td><td>The coprocessor number 0..15.</td></tr><tr><td><i>CRn</i></td><td>The coprocessor register to load.</td></tr><tr><td><i>src</i></td><td>A pointer to the data to load.</td></tr></table>	<i>coproc</i>	The coprocessor number 0..15.	<i>CRn</i>	The coprocessor register to load.	<i>src</i>	A pointer to the data to load.
<i>coproc</i>	The coprocessor number 0..15.						
<i>CRn</i>	The coprocessor register to load.						
<i>src</i>	A pointer to the data to load.						
Description	<p>Inserts the coprocessor load instruction LDC—or one of its variants—which means that a value will be loaded into a coprocessor register. The parameters <i>coproc</i> and <i>CRn</i> will be encoded in the instruction and must therefore be constants.</p> <p>The intrinsic functions __LDC and __LDCL require architecture ARMv4 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.</p> <p>The intrinsic functions __LDC2 and __LDC2L require architecture ARMv5 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.</p>						

**__LDC_noidx
__LDCL_noidx
__LDC2_noidx
__LDC2L_noidx**

Syntax	<code>void __LDCxxx_noidx(__ul coproc, __ul CRn, __ul const *src, __ul option);</code>		
Parameters	<table><tr><td><i>coproc</i></td><td>The coprocessor number 0..15.</td></tr></table>	<i>coproc</i>	The coprocessor number 0..15.
<i>coproc</i>	The coprocessor number 0..15.		

	<i>CRn</i>	The coprocessor register to load.
	<i>src</i>	A pointer to the data to load.
	<i>option</i>	Additional coprocessor option 0..255.
Description	<p>Inserts the coprocessor load instruction LDC, or one of its variants. A value will be loaded into a coprocessor register. The parameters <i>coproc</i>, <i>CRn</i>, and <i>option</i> will be encoded in the instruction and must therefore be constants.</p> <p>The intrinsic functions <code>__LDC_nidx</code> and <code>__LDCL_nidx</code> require architecture ARMv4 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.</p> <p>The intrinsic functions <code>__LDC2_nidx</code> and <code>__LDC2L_nidx</code> require architecture ARMv5 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.</p>	

__LDREX

__LDREXB

__LDREXD

__LDREXH

Syntax	<pre>unsigned long __LDREX(unsigned long *); unsigned char __LDREXB(unsigned char *); unsigned long long __LDREXD(unsigned long long *); unsigned short __LDREXH(unsigned short *);</pre>	
Description	<p>Inserts the specified instruction.</p> <p>The <code>__LDREX</code> intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.</p> <p>The <code>__LDREXB</code> and the <code>__LDREXH</code> intrinsic functions require architecture ARMv6K or ARMv7 for ARM mode, and ARMv7 for Thumb mode.</p> <p>The <code>__LDREX</code> intrinsic function requires architecture ARMv6K or ARMv7 for ARM mode, and ARMv7 but not ARMv7-M for Thumb mode.</p>	

__MCR

Syntax	<pre>void __MCR(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul CRm, __ul opcode_2);</pre>	
Parameters	<i>coproc</i>	The coprocessor number 0..15.

	<code>opcode_1</code>	Coprocessor-specific operation code.
	<code>src</code>	The value to be written to the coprocessor.
	<code>CRn</code>	The coprocessor register to write to.
	<code>CRm</code>	Additional coprocessor register; set to zero if not used.
	<code>opcode_2</code>	Additional coprocessor-specific operation code; set to zero if not used.
Description	<p>Inserts a coprocessor write instruction (<code>MCR</code>). A value will be written to a coprocessor register. The parameters <code>coproc</code>, <code>opcode_1</code>, <code>CRn</code>, <code>CRm</code>, and <code>opcode_2</code> will be encoded in the <code>MCR</code> instruction operation code and must therefore be constants.</p> <p>This intrinsic function requires either ARM mode, or an ARMv6T2 or higher for Thumb mode.</p>	

__MRC

Syntax	<pre>unsigned long __MRC(__ul coproc, __ul opcode_1, __ul CRn, __ul CRm, __ul opcode_2);</pre>	
Parameters	<code>coproc</code>	The coprocessor number 0..15.
	<code>opcode_1</code>	Coprocessor-specific operation code.
	<code>CRn</code>	The coprocessor register to write to.
	<code>CRm</code>	Additional coprocessor register; set to zero if not used.
	<code>opcode_2</code>	Additional coprocessor-specific operation code; set to zero if not used.
Description	<p>Inserts a coprocessor read instruction (<code>MRC</code>). Returns the value of the specified coprocessor register. The parameters <code>coproc</code>, <code>opcode_1</code>, <code>CRn</code>, <code>CRm</code>, and <code>opcode_2</code> will be encoded in the <code>MRC</code> instruction operation code and must therefore be constants.</p> <p>This intrinsic function requires either ARM mode, or an ARMv6T2 or higher for Thumb mode.</p>	

__no_operation

Syntax	<pre>void __no_operation(void);</pre>
Description	Inserts a <code>NOP</code> instruction.

__PKHBT

Syntax	unsigned long __PKHBT(unsigned long <i>x</i> , unsigned long <i>y</i> , unsigned long <i>count</i>);	
Parameters	<i>x</i>	First operand.
	<i>y</i>	Second operand, optionally shifted left.
	<i>count</i>	Shift count 0–31, where 0 means no shift.
Description	Inserts a <code>PKHBT</code> instruction, with an optionally shifted operand (LSL) for count in the range 1–31.	
	This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.	

__PKHTB

Syntax	unsigned long __PKHTB(unsigned long <i>x</i> , unsigned long <i>y</i> , unsigned long <i>count</i>);	
Parameters	<i>x</i>	First operand.
	<i>y</i>	Second operand, optionally shifted right (arithmetic shift).
	<i>count</i>	Shift count 0–32, where 0 means no shift.
Description	Inserts a <code>PKHTB</code> instruction, with an optionally shifted operand (ASR) for count in the range 1–32.	
	This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARM v7E-M for Thumb mode.	

__QADD
__QDADD
__QDSUB
__QSUB

Syntax	signed long __Qxxx(signed long, signed long);
Description	Inserts the specified instruction.

These intrinsic functions require architecture ARMv5E or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__QADD8
__QADD16
__QASX
__QSAX
__QSUB8
__QSUB16

Syntax	<code>unsigned long __Qxxx(unsigned long, unsigned long);</code>
Description	Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__QCFlag

Syntax	<code>unsigned long __QCFlag(void);</code>
Description	Returns the value of the cumulative saturation flag QC of the FPSCR register (Floating-point Status and Control Register). This intrinsic function is only available for devices with Neon (Advanced SIMD).

__QDOUBLE

Syntax	<code>signed long __QDOUBLE(signed long);</code>
Description	Inserts an instruction <code>QADD Rd, Rs, Rs</code> for a source register <code>Rs</code> , and a destination register <code>Rd</code> . This intrinsic function requires architecture ARMv5E or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__QFlag

Syntax	<code>int __QFlag(void);</code>
Description	Returns the Q flag that indicates if overflow/saturation has occurred.

This intrinsic function requires architecture ARMv5E or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__RBIT

Syntax	<code>unsigned long __RBIT(unsigned long);</code>
Description	Inserts an <code>RBIT</code> instruction, which reverses the bit order in a 32-bit register. This intrinsic function requires architecture ARMv6T2 or higher.

__reset_Q_flag

Syntax	<code>void __reset_Q_flag(void);</code>
Description	Clears the <code>Q</code> flag that indicates if overflow/saturation has occurred. This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__reset_QC_flag

Syntax	<code>void __reset_QC_flag(void);</code>
Description	Clears the value of the cumulative saturation flag <code>QC</code> of the <code>FPSCR</code> register (Floating-point Status and Control Register). This intrinsic function is only available for devices with Neon (Advanced SIMD).

__REV **__REV16** **__REVSH**

Syntax	<code>unsigned long __REV(unsigned long);</code> <code>unsigned long __REV16(unsigned long);</code> <code>signed long __REVSH(short);</code>
Description	Inserts the specified instruction. These intrinsic functions require architecture ARMv6 or higher.

__SADD8
__SADDI6
__SASX
__SSAX
__SSUB8
__SSUBI6

Syntax `unsigned long __Sxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SEL

Syntax `unsigned long __SEL(unsigned long, unsigned long);`

Description Inserts an `SEL` instruction.

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__set_BASEPRI

Syntax `void __set_BASEPRI(unsigned long);`

Description Sets the value of the `BASEPRI` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 or Cortex-M4 device.

__set_CONTROL

Syntax `void __set_CONTROL(unsigned long);`

Description Sets the value of the `CONTROL` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__set_CPSR

Syntax	<code>void __set_CPSR(unsigned long);</code>
Description	Sets the value of the ARM CPSR (Current Program Status Register). Only the control field is changed (bits 0-7). This intrinsic function can only be used in privileged mode, is not available for Cortex-M devices, and it requires ARM mode.

__set_FAULTMASK

Syntax	<code>void __set_FAULTMASK(unsigned long);</code>
Description	Sets the value of the FAULTMASK register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 or Cortex-M4 device.

__set_FPSCR

Syntax	<code>void __set_FPSCR(unsigned long);</code>
Description	Sets the value of FPSCR (floating-point status and control register) This intrinsic function is only available for devices with a VFP coprocessor.

__set_interrupt_state

Syntax	<code>void __set_interrupt_state(__istate_t);</code>
Descriptions	Restores the interrupt state to a value previously returned by the <code>__get_interrupt_state</code> function. For information about the <code>__istate_t</code> type, see <i>__get_interrupt_state</i> , page 349.

__set_LR

Syntax	<code>void __set_LR(unsigned long);</code>
Description	Assigns a new address to the link register (R14).

__set_MSP

Syntax

```
void __set_MSP(unsigned long);
```

Description

Sets the value of the `MSP` register (Main Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

__set_PRIMASK

Syntax

```
void __set_PRIMASK(unsigned long);
```

Description

Sets the value of the `PRIMASK` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__set_PSP

Syntax

```
void __set_PSP(unsigned long);
```

Description

Sets the value of the `PSP` register (Process Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

__set_SB

Syntax

```
void __set_SB(unsigned long);
```

Description

Assigns a new address to the static base register (`R9`).

__set_SP

Syntax

```
void __set_SP(unsigned long);
```

Description

Assigns a new address to the stack pointer register (`R13`).

__SEV

Syntax

```
void __SEV(void);
```

Description

Inserts an `SEV` instruction.

This intrinsic function requires architecture ARMv7 for ARM mode, and ARMv6-M or ARMv7 for Thumb mode.

__SHADD8
__SHADDI6
__SHASX
__SHSAX
__SHSUB8
__SHSUBI6

Syntax `unsigned long __SHxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SMLABB
__SMLABT
__SMLATB
__SMLATT
__SMLAWB
__SMLAWT

Syntax `unsigned long __SMLAxxx(unsigned long, unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SMLAD
__SMLADX
__SMLSD
__SMLSDX

Syntax

```
unsigned long __SMLxxx(unsigned long, unsigned long, unsigned
long);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SMLALBB
__SMLALBT
__SMLALTB
__SMLALTT

Syntax

```
unsigned long long __SMLALxxx(unsigned long, unsigned long,
unsigned long long);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SMLALD
__SMLALDX
__SMLSLD
__SMLSLDX

Syntax

```
unsigned long long __SMLxxx(unsigned long, unsigned long,
unsigned long long);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SMMLA **__SMMLAR** **__SMMLS** **__SMMLSR**

Syntax	<code>unsigned long __SMMLxxx(unsigned long, unsigned long, unsigned long);</code>
Description	<p>Inserts the specified instruction.</p> <p>These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.</p>

__SMMUL **__SMMULR**

Syntax	<code>unsigned long __SMMULxxx(unsigned long, unsigned long);</code>
Description	<p>Inserts the specified instruction.</p> <p>These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.</p>

__SMUAD **__SMUADX** **__SMUSD** **__SMUSDX**

Syntax	<code>unsigned long __SMUxxx(unsigned long, unsigned long);</code>
Description	<p>Inserts the specified instruction.</p> <p>These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.</p>

__SMUL

Syntax	<code>signed long __SMUL(signed short, signed short);</code>
Description	Inserts a signed 16-bit multiplication.

This intrinsic function requires architecture ARMv5-E or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SMULBB
__SMULBT
__SMULTB
__SMULTT
__SMULWB
__SMULWT

Syntax `unsigned long __SMULxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SSAT

Syntax `unsigned long __SSAT(unsigned long, unsigned long);`

Description Inserts an `SSAT` instruction.

The compiler will incorporate a shift instruction into the operand when possible. For example, `__SSAT(x << 3, 11)` compiles to `SSAT Rd, #11, Rn, LSL #3`, where the value of `x` has been placed in register `Rn` and the return value of `__SSAT` will be placed in register `Rd`.

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__SSAT16

Syntax `unsigned long __SSAT16(unsigned long, unsigned long);`

Description Inserts an `SSAT16` instruction.

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARM v7E-M for Thumb mode.

__STC
__STCL
__STC2
__STC2L

Syntax `void __STCxxx(__ul coproc, __ul CRn, __ul const *dst);`

Parameters

<i>coproc</i>	The coprocessor number 0..15.
<i>CRn</i>	The coprocessor register to load.
<i>dst</i>	A pointer to the destination.

Description

Inserts the coprocessor store instruction *STC*—or one of its variants—which means that the value of the specified coprocessor register will be written to a memory location. The parameters *coproc* and *CRn* will be encoded in the instruction and must therefore be constants.

The intrinsic functions `__STC` and `__STCL` require architecture ARMv4 or higher for ARM mode, and ARM v6T2 or higher for Thumb mode.

The intrinsic functions `__STC2` and `__STC2L` require architecture ARMv5 or higher for ARM mode, and ARMv6-T2 or higher for Thumb mode.

__STC_noidx
__STCL_noidx
__STC2_noidx
__STC2L_noidx

Syntax `void __STCxxx_noidx(__ul coproc, __ul CRn, __ul const *dst, __ul option);`

Parameters

<i>coproc</i>	The coprocessor number 0..15.
<i>CRn</i>	The coprocessor register to load.
<i>dst</i>	A pointer to the destination.
<i>option</i>	Additional coprocessor option 0..255.

Description

Inserts the coprocessor store instruction *STC*—or one of its variants—which means that the value of the specified coprocessor register will be written to a memory location. The parameters *coproc*, *CRn*, and *option* will be encoded in the instruction and must therefore be constants.

The intrinsic functions `__STC_noidx` and `__STCL_noidx` require architecture ARMv4 or higher for ARM mode, and ARMv6-T2 or higher for Thumb mode.

The intrinsic functions `__STC2_noidx` and `__STC2L_noidx` require architecture ARMv5 or higher for ARM mode, and ARMv6-T2 or higher for Thumb mode.

__STREX
__STREXB
__STREXD
__STREXH

Syntax

```
unsigned long __STREX(unsigned long, unsigned long *);
unsigned long __STREXB(unsigned char, unsigned char *);
unsigned long __STREXD(unsigned long long, unsigned long long *);
unsigned long __STREXH(unsigned short, unsigned short *);
```

Description

Inserts the specified instruction.

The `__STREX` intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv6-T2 or higher for Thumb mode.

The `__STREXB` and the `__STREXH` intrinsic functions require architecture ARMv6K or ARMv7 for ARM mode, and ARMv7 for Thumb mode.

The `__STREXD` intrinsic function requires architecture ARMv6K or ARMv7 for ARM mode, and ARMv7 except for ARMv7-M for Thumb mode.

__SWP
__SWPB

Syntax

```
unsigned long __SWP(unsigned long, unsigned long *);
char __SWPB(unsigned char, unsigned char *);
```

Description

Inserts the specified instruction.

These intrinsic functions require ARM mode.

__SXTAB
__SXTAB16
__SXTAH
__SXTB16

Syntax `unsigned long __SXTxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__UADD8
__UADD16
__UASX
__USAX
__USUB8
__USUB16

Syntax `unsigned long __Uxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__UHADD8
__UHADD16
__UHASX
__UHSAX
__UHSUB8
__UHSUB16

Syntax `unsigned long __UHxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__UMAAL

Syntax `unsigned long long __UMAAL(unsigned long, unsigned long, unsigned long, unsigned long);`

Description Inserts an `UMAAL` instruction.

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__UQADD8 **__UQADD16** **__UQASX** **__UQSAX** **__UQSUB8** **__UQSUB16**

Syntax `unsigned long __UQxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__USAD8 **__USADA8**

Syntax `unsigned long __USADxxx(unsigned long, unsigned long);`

Description Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__USAT

Syntax `unsigned long __USAT(unsigned long, unsigned long);`

Description Inserts a `USAT` instruction.

The compiler will incorporate a shift instruction into the operand when possible. For example, `__USAT(x << 3, 11)` compiles to `USAT Rd, #11, Rn, LSL #3`, where the value of `x` has been placed in register `Rn` and the return value of `__USAT` will be placed in register `Rd`.

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7-M for Thumb mode.

__USAT16

Syntax

```
unsigned long __USAT16(unsigned long, unsigned long);
```

Description

Inserts a `USAT16` instruction.

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__UXTAB **__UXTAB16** **__UXTAH** **__UXTB16**

Syntax

```
unsigned long __UXTxxx(unsigned long, unsigned long);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.

__WFI **__WFE** **__YIELD**

Syntax

```
void long __xxx(void);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture ARMv7 for ARM mode, and ARMv6-M, or ARMv7 for Thumb mode.

The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for ARM adheres to Standard C. The compiler also makes these preprocessor-related features available to you:

- **Predefined preprocessor symbols**
These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For more information, see *Descriptions of predefined preprocessor symbols*, page 368.
- **User-defined preprocessor symbols defined using a compiler option**
In addition to defining your own preprocessor symbols using the `#define` directive, you can also use the option `-D`, see *-D*, page 244.
- **Preprocessor extensions**
There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives*. For information about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 371.
- **Preprocessor output**
Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 266.

To specify a path for an include file, use forward slashes:

```
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile", "rt");
```

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.

Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

Predefined symbol	Identifies
__AAPCS__	An integer that is set based on the <code>--aapcs</code> option. The symbol is set to 1 if the AAPCS base standard is the selected calling convention (<code>--aapcs=std</code>). The symbol is undefined for other calling conventions.
__AAPCS_VFP__	An integer that is set based on the <code>--aapcs</code> option. The symbol is set to 1 if the VFP variant of AAPCS is the selected calling convention (<code>--aapcs=vfp</code>). The symbol is undefined for other calling conventions.
__ARM_ADVANCED_SIMD__	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the Advanced SIMD architecture extension. The symbol is undefined for other cores.
__ARM_MEDIA__	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the ARMv6 SIMD extensions for multimedia. The symbol is undefined for other cores.
__ARM_PROFILE_M__	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture is a profile M core. The symbol is undefined for other cores.
__ARMVFP__	An integer that reflects the <code>--fpu</code> option and is defined to <code>__ARMVFPV2__</code> , <code>__ARMVFPV3__</code> , or <code>__ARMVFPV4__</code> . These symbolic names can be used when testing the <code>__ARMVFP__</code> symbol. If VFP code generation is disabled (default), the symbol will be undefined.
__ARMVFP_D16__	An integer that is set based on the <code>--fpu</code> option. The symbol is set to 1 if the selected FPU is a VFPv3 or VFPv4 unit with only 16 D registers. Otherwise, the symbol is undefined.
__ARMVFP_FP16__	An integer that is set based on the <code>--fpu</code> option. The symbol is set to 1 if the selected FPU only supports 16-bit floating-point numbers. Otherwise, the symbol is undefined.
__ARMVFP_SP__	An integer that is set based on the <code>--fpu</code> option. The symbol is set to 1 if the selected FPU only supports single-precision. Otherwise, the symbol is undefined.

Table 36: Predefined symbols

Predefined symbol	Identifies
<code>__BASE_FILE__</code>	A string that identifies the name of the base source file (that is, not the header file), being compiled. See also <code>__FILE__</code> , page 369, and <code>--no_path_in_file_macros</code> , page 260.
<code>__BUILD_NUMBER__</code>	A unique integer that identifies the build number of the compiler currently in use.
<code>__CORE__</code>	An integer that identifies the processor architecture in use. The symbol reflects the <code>--core</code> option and is defined to <code>__ARM4M__</code> , <code>__ARM4TM__</code> , <code>__ARM5__</code> , <code>__ARM5E__</code> , <code>__ARM6__</code> , <code>__ARM6M__</code> , <code>__ARM6SM__</code> , <code>__ARM7M__</code> , <code>__ARM7EM__</code> , <code>__ARM7A__</code> , or <code>__ARM7R__</code> . These symbolic names can be used when testing the <code>__CORE__</code> symbol.
<code>__cplusplus</code>	An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
<code>__CPU_MODE__</code>	An integer that reflects the selected CPU mode and is defined to 1 for Thumb and 2 for ARM.
<code>__DATE__</code>	A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Oct 30 2010".*
<code>__DOUBLE__</code>	An integer that identifies the size of the data type <code>double</code> . The symbol is defined to 64.
<code>__embedded_cplusplus</code>	An integer which is defined to 1 when the compiler runs in Embedded C++ or Extended Embedded C++ mode, otherwise the symbol is undefined. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
<code>__FILE__</code>	A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also <code>__BASE_FILE__</code> , page 369, and <code>--no_path_in_file_macros</code> , page 260.*

Table 36: Predefined symbols (Continued)

Predefined symbol	Identifies
<code>__func__</code>	A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <i>-e</i> , page 249. See also <code>__PRETTY_FUNCTION__</code> , page 370.*
<code>__FUNCTION__</code>	A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <i>-e</i> , page 249. See also <code>__PRETTY_FUNCTION__</code> , page 370.
<code>__IAR_SYSTEMS_ICC__</code>	An integer that identifies the IAR compiler platform. The current value is 8. Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was compiled by a compiler from IAR Systems.
<code>__ICCARM__</code>	An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for ARM.
<code>__LINE__</code>	An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.*
<code>__LITTLE_ENDIAN__</code>	An integer that reflects the <code>--endian</code> option and is defined to 1 when the byte order is little-endian. The symbol is defined to 0 when the byte order is big-endian.
<code>__PRETTY_FUNCTION__</code>	A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example <code>"void func(char) "</code> . This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <i>-e</i> , page 249. See also <code>__func__</code> , page 370.
<code>__ROPI__</code>	An integer that is defined when the <code>--ropi</code> compiler option is used; see <i>-ropi</i> , page 254.
<code>__RWPI__</code>	An integer that is defined when the <code>--rwpi</code> compiler option is used; see <i>-rwpi</i> , page 255.
<code>__STDC__</code>	An integer that is set to 1, which means the compiler adheres to Standard C. This symbol can be tested with <code>#ifdef</code> to detect whether the compiler in use adheres to Standard C.*

Table 36: Predefined symbols (Continued)

Predefined symbol	Identifies
__STDC_VERSION__	An integer that identifies the version of the C standard in use. The symbol expands to 199901L, unless the -c89 compiler option is used in which case the symbol expands to 199409L. This symbol does not apply in EC++ mode.*
__TIME__	A string that identifies the time of compilation in the form "hh:mm:ss".*
__VER__	An integer that identifies the version number of the IAR compiler in use. For example, version 5.11.3 is returned as 5011003.

Table 36: Predefined symbols (Continued)

* This symbol is required by Standard C.

Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and Standard C directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

See also

Assert, page 136.



In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

#warning message

Syntax

`#warning message`

where *message* can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.

Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

Library overview

The compiler comes with the IAR DLIB Library, a complete library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.

For more information about customization, see the chapter *The DLIB runtime environment*.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For more information about library functions, see the chapter *Implementation-defined behavior for Standard C* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to set up a runtime library, see *Setting up the runtime environment, page 94*. The linker will include only those routines that are required—directly or indirectly—by your application.

ALTERNATIVE MORE ACCURATE LIBRARY FUNCTIONS

The default implementation of `cos`, `sin`, `tan`, and `pow` is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named `__iar_xxx_accuratef` for float variants of the functions and `__iar_xxx_accuratel` for long double variants of the functions, and where `xxx` is `cos`, `sin`, etc.

To use any of these more accurate versions, use the `--redirect` linker option.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `printf`, `sprintf`, `scanf`, `sscanf`, `getchar`, and `putchar`.

Functions that can set `errno` are not reentrant, because an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. This applies to math and string conversion functions, among others.

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines

- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

THE LONGJMP FUNCTION



A `longjmp` is in effect a jump to a previously defined `setjmp`. Any variable length arrays or C++ objects residing on the stack during stack unwinding will not be destroyed. This can lead to resource leaks or incorrect application behavior.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code, see the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of ARM features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, see *Added C functionality*, page 379.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Using C*.

This table lists the C header files:

Header file	Usage
<code>assert.h</code>	Enforcing assertions when functions execute
<code>complex.h</code>	Computing common complex mathematical functions
<code>ctype.h</code>	Classifying characters
<code>errno.h</code>	Testing error codes reported by library functions

Table 37: Traditional Standard C header files—DLIB

Header file	Usage
<code>fenv.h</code>	Floating-point exception flags
<code>float.h</code>	Testing floating-point type properties
<code>inttypes.h</code>	Defining formatters for all types defined in <code>stdint.h</code>
<code>iso646.h</code>	Using Amendment 1— <code>iso646.h</code> standard header
<code>limits.h</code>	Testing integer type properties
<code>locale.h</code>	Adapting to different cultural conventions
<code>math.h</code>	Computing common mathematical functions
<code>setjmp.h</code>	Executing non-local goto statements
<code>signal.h</code>	Controlling various exceptional conditions
<code>stdarg.h</code>	Accessing a varying number of arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C.
<code>stddef.h</code>	Defining several useful types and macros
<code>stdint.h</code>	Providing integer characteristics
<code>stdio.h</code>	Performing input and output
<code>stdlib.h</code>	Performing a variety of operations
<code>string.h</code>	Manipulating several kinds of strings
<code>tgmath.h</code>	Type-generic mathematical functions
<code>time.h</code>	Converting between various time and date formats
<code>uchar.h</code>	Unicode functionality (IAR extension to Standard C)
<code>wchar.h</code>	Support for wide characters
<code>wctype.h</code>	Classifying wide characters

Table 37: Traditional Standard C header files—DLIB (Continued)

C++ HEADER FILES

This section lists the C++ header files:

- The C++ library header files
The header files that constitute the Standard C++ and the Embedded C++ library.
- The C++ standard template library (STL) header files
The header files that constitute STL for the Standard C++ and the Extended Embedded C++ library.
- The C++ C header files
The C++ header files that provide the resources from the C library.

The C++ library header files

This table lists the header files that can be used in C++ as well as in Embedded C++:

Header file	Usage
<code>complex</code>	Defining a class that supports complex arithmetic
<code>exception</code>	Defining several functions that control exception handling; only usable in C++
<code>fstream</code>	Defining several I/O stream classes that manipulate external files
<code>iomanip</code>	Declaring several I/O stream manipulators that take an argument
<code>ios</code>	Defining the class that serves as the base for many I/O streams classes
<code>iosfwd</code>	Declaring several I/O stream classes before they are necessarily defined
<code>iostream</code>	Declaring the I/O stream objects that manipulate the standard streams
<code>istream</code>	Defining the class that performs extractions
<code>limits</code>	Defining numerical values; only usable in C++
<code>locale</code>	Adapting to different cultural conventions; only usable in C++
<code>new</code>	Declaring several functions that allocate and free storage
<code>ostream</code>	Defining the class that performs insertions
<code>sstream</code>	Defining several I/O stream classes that manipulate string containers
<code>stdexcept</code>	Defining several classes useful for reporting exceptions; only usable in C++
<code>streambuf</code>	Defining classes that buffer I/O stream operations
<code>string</code>	Defining a class that implements a string container
<code>stringstream</code>	Defining several I/O stream classes that manipulate in-memory character sequences
<code>typeinfo</code>	Defining type information support; only usable in C++

Table 38: C++ header files

The C++ standard template library (STL) header files

The following table lists the standard template library (STL) header files that can be used in C++ as well as in Extended Embedded C++:

Header file	Description
<code>algorithm</code>	Defines several common operations on sequences
<code>bitset</code>	Defining a container with fixed-sized sequences of bits; only usable in C++
<code>deque</code>	A deque sequence container
<code>functional</code>	Defines several function objects

Table 39: <Standard template library header files

Header file	Description
hash_map	A map associative container, based on a hash algorithm
hash_set	A set associative container, based on a hash algorithm
iterator	Defines common iterators, and operations on iterators
list	A doubly-linked list sequence container
map	A map associative container
memory	Defines facilities for managing memory
numeric	Performs generalized numeric operations on sequences
queue	A queue sequence container
set	A set associative container
slist	A singly-linked list sequence container
stack	A stack sequence container
utility	Defines several utility components
valarray	Defining varying length array container; only usable in C++
vector	A vector sequence container

Table 39: <Standard template library header files (Continued)

Using Standard C libraries in C++

The C++ library works in conjunction with some of the header files from the Standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

This table shows the new header files:

Header file	Usage
cassert	Enforcing assertions when functions execute
ccomplex	Computing common complex mathematical functions
cctype	Classifying characters
cerrno	Testing error codes reported by library functions
cfenv	Floating-point exception flags
cfloat	Testing floating-point type properties
cinttypes	Defining formatters for all types defined in <code>stdint.h</code>
ciso646	Using Amendment 1— <code>iso646.h</code> standard header
climits	Testing integer type properties
locale	Adapting to different cultural conventions

Table 40: New Standard C header files—DLIB

Header file	Usage
<code>cmath</code>	Computing common mathematical functions
<code>csetjmp</code>	Executing non-local goto statements
<code>csignal</code>	Controlling various exceptional conditions
<code>cstdarg</code>	Accessing a varying number of arguments
<code>cstdbool</code>	Adds support for the <code>bool</code> data type in C.
<code>cstddef</code>	Defining several useful types and macros
<code>cstdint</code>	Providing integer characteristics
<code>cstdio</code>	Performing input and output
<code>cstdlib</code>	Performing a variety of operations
<code>cstring</code>	Manipulating several kinds of strings
<code>ctgmath</code>	Type-generic mathematical functions
<code>ctime</code>	Converting between various time and date formats
<code>cwchar</code>	Support for wide characters
<code>cwctype</code>	Classifying wide characters

Table 40: New Standard C header files—DLIB (Continued)

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality.

The following include files provide these features:

- `fenv.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

fenv.h

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapenable` and `fegettrapdisable`.

stdio.h

These functions provide additional I/O functionality:

<code>fdopen</code>	Opens a file based on a low-level file descriptor.
<code>fileno</code>	Gets the low-level file descriptor from the file descriptor (<code>FILE*</code>).
<code>__gets</code>	Corresponds to <code>fgets</code> on <code>stdin</code> .
<code>getw</code>	Gets a <code>wchar_t</code> character from <code>stdin</code> .
<code>putw</code>	Puts a <code>wchar_t</code> character to <code>stdout</code> .
<code>__ungetchar</code>	Corresponds to <code>ungetc</code> on <code>stdout</code> .
<code>__write_array</code>	Corresponds to <code>fwrite</code> on <code>stdout</code> .

string.h

These are the additional functions defined in `string.h`:

<code>strdup</code>	Duplicates a string on the heap.
<code>strcasemp</code>	Compares strings case-insensitive.
<code>strncasemp</code>	Compares strings case-insensitive and bounded.
<code>strlen</code>	Bounded string length.

time.h

There are two interfaces for using `time_t` and the associated functions `time`, `ctime`, `difftime`, `gmtime`, `localtime`, and `mktime`:

- The 32-bit interface supports years from 1900 up to 2035 and uses a 32-bit integer for `time_t`. The type and function have names like `__time32_t`, `__time32`, etc. This variant is mainly available for backwards compatibility.
- The 64-bit interface supports years from -9999 up to 9999 and uses a signed `long long` for `time_t`. The type and function have names like `__time64_t`, `__time64`, etc.

The interfaces are defined in the system header file `time.h`.

An application can use either interface, and even mix them by explicitly using the 32- or 64-bit variants. By default, the library and the header redirect `time_t`, `time` etc. to

the 32-bit variants. However, to explicitly redirect them to their 64-bit variants, define `_DLIB_TIME_USES_64` in front of the inclusion of `time.h` or `ctime`.

See also, *Time*, page 121.

SYMBOLS USED INTERNALLY BY THE LIBRARY

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__code, __data`

These symbols are used as memory attributes internally by the compiler, and they might have to be used as arguments in certain templates.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor, __has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

Note: The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

The linker configuration file

This chapter describes the purpose of the linker configuration file and describes its contents.

To read this chapter you must be familiar with the concept of *sections*, see *Modules and sections*, page 82.

Overview

To link and locate an application in memory according to your requirements, ILINK needs information about how to handle sections and how to place them into the available memory regions. In other words, ILINK needs a *configuration*, passed to it by means of the *linker configuration file*.

This file consists of a sequence of directives and typically, provides facilities for:

- Defining available addressable memories
 - giving the linker information about the maximum size of possible addresses and defining the available physical memory, as well as dealing with memories that can be addressed in different ways.
- Defining the regions of the available memories that are populated with ROM or RAM
 - giving the start and end address for each region.
- Section groups
 - dealing with how to group sections into blocks and overlays depending on the section requirements.
- Defining how to handle initialization of the application
 - giving information about which sections that are to be initialized, and how that initialization should be made.
- Memory allocation
 - defining where—in what memory region—each set of sections should be placed.
- Using symbols, expressions, and numbers
 - expressing addresses and sizes, etc, in the other configuration directives. The symbols can also be used in the application itself.

- Structural configuration
meaning that you can include or exclude directives depending on a condition, and to split the configuration file into several different files.

Comments can be written either as C comments (`/* . . . */`) or as C++ comments (`// . . .`).

Defining memories and regions

ILINK needs information about the available memory spaces, or more specifically it needs information about:

- The maximum size of possible addressable memories
The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. See *Define memory directive*, page 384.
- Available physical memory
The `define region` directive defines a region in the available memories in which specific sections of application code and sections of application data can be placed. See *Define region directive*, page 385.
A region consists of one or several memory ranges. A range is a continuous sequence of bytes in a memory and several ranges can be expressed by using region expressions. See *Region expression*, page 387.

Define memory directive

Syntax

```
define memory [ name ] with size = size_expr [ ,unit-size ] ;
```

where *unit-size* is one of:

```
unitbitsize = bitsize_expr  
unitbytesize = bytesize_expr
```

and where *expr* is an expression, see *Expressions*, page 401.

Parameters	<i>size_expr</i>	Specifies how many <i>units</i> the memory space contains; always counted from address zero.
	<i>bitsize_expr</i>	Specifies how many bits each unit contains.
	<i>bytesize_expr</i>	Specifies how many bytes each unit contains. Each byte contains 8 bits.

Description	The <code>define memory</code> directive defines a <i>memory space</i> with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. This sets the limits for the possible addresses to be used in the linker configuration file. For many microcontrollers, one memory space is sufficient. However, some microcontrollers require two or more. For example, a Harvard architecture usually requires two different memory spaces, one for code and one for data. If only one memory is defined, the memory name is optional. If no <i>unit-size</i> is given, the unit contains 8 bits.
Example	<pre>/* Declare the memory space Mem of four Gigabytes */ define memory Mem with size = 4G;</pre>

Define region directive

Syntax	<pre>define region name = region-expr;</pre> <p>where <i>region-expr</i> is a region expression, see also <i>Regions</i>, page 385.</p>		
Parameters	<table><tr><td><i>name</i></td><td>The name of the region.</td></tr></table>	<i>name</i>	The name of the region.
<i>name</i>	The name of the region.		
Description	The <code>define region</code> directive defines a region in which specific sections of code and sections of data can be placed. A region consists of one or several memory ranges, where each memory range consists of a continuous sequence of bytes in a specific memory. Several ranges can be combined by using region expressions. Note that those ranges do not need to be consecutive or even in the same memory.		
Example	<pre>/* Define the 0x10000-byte code region ROM located at address 0x10000 in memory Mem */ define region ROM = Mem:[from 0x10000 size 0x10000];</pre>		

Regions

A *region* is a set of non-overlapping memory ranges. A *region expression* is built up out of *region literals* and set operations (union, intersection, and difference) on regions.

Region literal

Syntax	<pre>[memory-name:][from expr { to expr size expr } [repeat expr [displacement expr]]]</pre>
--------	---

where *expr* is an expression, see *Expressions*, page 401.

Parameters

<i>memory-name</i>	The name of the memory space in which the region literal will be located. If there is only one memory, the name is optional.
<i>from expr</i>	<i>expr</i> is the start address of the memory range (inclusive).
<i>to expr</i>	<i>expr</i> is the end address of the memory range (inclusive).
<i>size expr</i>	<i>expr</i> is the size of the memory range.
<i>repeat expr</i>	<i>expr</i> defines several ranges in the same memory for the region literal.
<i>displacement expr</i>	<i>expr</i> is the displacement from the previous range start in the repeat sequence. Default displacement is the same value as the range size.

Description

A region literal consists of one memory range. When you define a range, the memory it resides in, a start address, and a size must be specified. The range size can be stated explicitly by specifying a size, or implicitly by specifying the final address of the range. The final address is included in the range and a zero-sized range will only contain an address. A range can span over the address zero and such a range can even be expressed by unsigned values, because it is known where the memory wraps.

The *repeat* parameter will create a region literal that contains several ranges, one for each repeat. This is useful for *banked* or *far* regions.

Example

```
/* The 5-byte size range spans over the address zero */
Mem:[from -2 to 2]

/* The 512-byte size range spans over zero, in a 64-Kbyte memory
*/
Mem:[from 0xFF00 to 0xFF]

/* Defining several ranges in the same memory, a repeating
literal */
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]

/* Resulting in a region containing:
Mem:[from 0 size 0x100]
Mem:[from 0x1000 size 0x100]
Mem:[from 0x2000 size 0x100]
*/
```

See also

Define region directive, page 385, and *Region expression*, page 387.

Region expression

Syntax

```
region-operand
| region-expr | region-operand
| region-expr - region-operand
| region-expr & region-operand
```

where *region-operand* is one of:

```
( region-expr )
region-name
region-literal
empty-region
```

where *region-name* is a region, see *Define region directive*, page 385

where *region-literal* is a region literal, see *Region literal*, page 385

and where *empty-region* is an empty region, see *Empty region*, page 388.

Description

Normally, a region consists of one memory range, which means a *region literal* is sufficient to express it. When a region contains several ranges, possibly in different memories, it is instead necessary to use a *region expression* to express it. Region expressions are actually set expressions on sets of memory ranges.

To create region expressions, three operators are available: union (`|`), intersection (`&`), and difference (`-`). These operators work as in *set theory*. For example, if you have the sets A and B, then the result of the operators would be:

- A | B: all elements in either set A or set B
- A & B: all elements in both set A and B
- A - B: all elements in set A but not in B.

Example

```
/* Resulting in a range starting at 1000 and ending at 2FFF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1500 and ending at 1FFF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1000 and ending at 14FF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]
```

```
/* Resulting in two ranges. The first starting at 1000 and ending
   at 1FFF, the second starting at 2501 and ending at 2FFF.
   Both located in memory Mem */
Mem:[from 0x1000 to 0x2FFF] - Mem:[from 0x2000 to 0x24FF]
```

Empty region

Syntax	[]
Description	The empty region does not contain any memory ranges. If the empty region is used in a placement directive that actually is used for placing one or more sections, ILINK will issue an error.
Example	<pre>define region Code = Mem:[from 0 size 0x10000]; if (Banked) { define region Bank = Mem:[from 0x8000 size 0x1000]; } else { define region Bank = []; } define region NonBanked = Code - Bank; /* Depending on the Banked symbol, the NonBanked region is either one range with 0x10000 bytes, or two ranges with 0x8000 and 0x7000 bytes, respectively. */</pre>
See also	<i>Region expression, page 387.</i>

Section handling

Section handling describes how ILINK should handle the sections of the execution image, which means:

- **Placing sections in regions**
The `place at` and `place into` directives place sets of sections with similar attributes into previously defined regions. See *Place at directive, page 395* and *Place in directive, page 396*.
- **Making sets of sections with special requirements**
The `block` directive makes it possible to create empty sections with specific sizes and alignments, sequentially sorted sections of different types, etc.
The `overlay` directive makes it possible to create an area of memory that can contain several overlay images. See *Define block directive, page 389*, and *Define overlay directive, page 390*.

- Initializing the application

The directives `initialize` and `do not initialize` control how the application should be started. With these directives, the application can initialize global symbols at startup, and copy pieces of code. The initializers can be stored in several ways, for example they can be compressed. See *Initialize directive, page 392* and *Do not initialize directive, page 394*.
- Keeping removed sections

The `keep` directive retains sections even though they are not referred to by the rest of the application, which means it is equivalent to the *root* concept in the assembler and compiler. See *Keep directive, page 394*.

Define block directive

Syntax

```
define [movable] block name
    [ with param, param... ]
{
    extended-selectors
}
[except
{
    section_selectors
}];
```

where *param* can be one of:

```
size = expr
maximum size = expr
alignment = expr
fixed order
static base [name]
static base
```

and where the rest of the directive selects sections to include in the block, see *Section selection, page 396*.

Parameters

<i>name</i>	The name of the block to be defined.
<i>size</i>	Customizes the size of the block. By default, the size of a block is the sum of its parts dependent of its contents.
<i>maximum size</i>	Specifies an upper limit for the size of the block. An error is generated if the sections in the block do not fit.

	<code>alignment</code>	Specifies a minimum alignment for the block. If any section in the block has a higher alignment than the minimum alignment, the block will have that alignment.
	<code>fixed order</code>	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.
	<code>static base name</code>	Specifies that the static base with the specified name will be at the start of the block when the application executes.
	<code>static base</code>	Specifies that the static base will be at the start of the block when the application executes.
Description	<p>The <code>block</code> directive defines a named set of sections. By defining a block you can create empty blocks of bytes that can be used, for example as stacks or heaps. Another use for the directive is to group certain types of sections, consecutive or non-consecutive. A third example of use for the directive is to group sections into one memory area to access the start and end of that area from the application.</p> <p><code>movable</code> blocks are for use with read-only and read-write position independence. Making blocks movable enables the linker to validate the application's use of addresses. Movable blocks are located in exactly the same way as other blocks, but the linker will check that the appropriate relocations are used when referring to symbols in movable blocks.</p>	
Example	<pre>/* Create a 0x1000-byte block for the heap */ define block HEAP with size = 0x1000, alignment = 8 { };</pre>	
See also	<i>Interaction between the tools and your application, page 190. See Define overlay directive, page 390 for an Accessing example.</i>	

Define overlay directive

Syntax	<pre>define overlay name [with param, param...] { extended-selectors; } [except { section_selectors }];</pre> <p>For information about extended selectors and except clauses, see <i>Section selection, page 396</i>.</p>
--------	---

Parameters

<code>name</code>	The name of the overlay.
<code>size</code>	Customizes the size of the overlay. By default, the size of a overlay is the sum of its parts dependent of its contents.
<code>maximum size</code>	Specifies an upper limit for the size of the overlay. An error is generated if the sections in the overlay do not fit.
<code>alignment</code>	Specifies a minimum alignment for the overlay. If any section in the overlay has a higher alignment than the minimum alignment, the overlay will have that alignment.
<code>fixed order</code>	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.

Description

The `overlay` directive defines a named set of sections. In contrast to the `block` directive, the `overlay` directive can define the same name several times. Each definition will then be grouped in memory at the same place as all other definitions of the same name. This creates an *overlaid* memory area, which can be useful for an application that has several independent sub-applications.

Place each sub-application image in ROM and reserve a RAM overlay area that can hold all sub-applications. To execute a sub-application, first copy it from ROM to the RAM overlay. Note that ILINK does not help you with managing interdependent overlay definitions, apart from generating a diagnostic message for any reference from one overlay to another overlay.

The size of an overlay will be the same size as the largest definition of that overlay name and the alignment requirements will be the same as for the definition with the highest alignment requirements.

Note: Sections that were overlaid must be split into a RAM and a ROM part and you must take care of all the copying needed.

See also

Manual initialization, page 101.

Initialize directive

Syntax

```
initialize { by copy | manually }
    [ with param, param... ]
{
    section-selectors
}
[except
{
    section_selectors
}];
```

where *param* is one of:

```
packing = { none | zeros | packbits | bwt | lzw | auto |
           smallest }
```

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 396.

Parameters

by copy	Splits the section into sections for initializers and initialized data, and handles the initialization at application startup automatically.
manually	Splits the section into sections for initializers and initialized data. The initialization at application startup is not handled automatically.
packing	<p>Specifies how to handle the initializers. Choose between:</p> <p><i>none</i> - Disables compression of the selected section contents. This is the default method for initialize manually.</p> <p><i>zeros</i> - Compresses sequential bytes with the value zero.</p> <p><i>packbits</i> - Compresses with the PackBits algorithm. This method generates good results for data with many consecutive bytes of the same value.</p> <p><i>bwt</i> - Compresses with the Burrows-Wheeler algorithm. This method improves the <i>packbits</i> method by transforming blocks of data before they are compressed.</p> <p><i>lzw</i> - Compresses with the Lempel-Ziv-Welch algorithm. This method uses a dictionary to store byte patterns in the data.</p> <p><i>auto</i> - Similar to <i>smallest</i>, but ILINK chooses between <i>none</i> and <i>packbits</i>. This is the default method for initialize by copy.</p> <p><i>smallest</i> - ILINK estimates the resulting size using each packing method (except for <i>auto</i>), and then chooses the packing method that produces the smallest estimated size. Note that the size of the decompressor is also included.</p>

Description

The `initialize` directive splits the initialization section into one section holding the initializers and another section holding the initialized data. You can choose whether the initialization at startup should be handled automatically (`initialize by copy`) or whether you should handle it yourself (`initialize manually`).

When you use the packing method `auto` (default for `initialize by copy`) or `smallest`, ILINK will automatically choose an appropriate packing algorithm for the initializers. To override this, specify a different packing method. The `--log initialization` option shows how ILINK decided which packing algorithm to use.

When initializers are compressed, a decompressor is automatically added to the image. The decompressors for `bwt` and `lzw` use significantly more execution time and RAM than `zeros` and `packbits`. Approximately 9 Kbytes of stack space is needed for `bwt` and 3.5 Kbytes for `lzw`.

When initializers are compressed, the exact size of the compressed initializers is unknown until the exact content of the uncompressed data is known. If this data contains other addresses, and some of these addresses are dependent on the size of the compressed initializers, the linker fails with error Lp017. To avoid this, place compressed initializers last, or in a memory region together with sections whose addresses do not need to be known.

Unless `initialize manually` is used, ILINK will arrange for initialization to occur during system startup by including an initialization table. Startup code calls an initialization routine that reads this table and performs the necessary initializations.

Zero-initialized sections are not affected by the `initialize` directive.

The `initialize` directive is normally used for initialized variables, but can be used for copying any sections, for example copying executable code from slow ROM to fast RAM, or for overlays. For another example, see *Define overlay directive, page 390*.

Sections that are needed for initialization are not affected by the `initialize by copy` directive. This includes the `__low_level_init` function and anything it references.

Anything reachable from the program entry label is considered *needed for initialization* unless reached via a section fragment with a label starting with `__iar_init$$done`. The `--log sections` option, in addition to logging the marking of section fragments to be included in the application, also logs the process of determining which sections are needed for initialization.

Example

```
/* Copy all read-write sections automatically from ROM to RAM at
   program start */
initialize by copy { rw };
place in RAM { rw };
place in ROM { ro };
```

See also *Initialization at system startup, page 88, and Do not initialize directive, page 394.*

Do not initialize directive

Syntax

```
do not initialize
{
    section-selectors
}
[except
{
    section-selectors
}];
```

For information about extended selectors and except clauses, see *Section selection, page 396*.

Description

The `do not initialize` directive specifies the sections that should not be initialized by the system startup code. The directive can only be used on `zeroinit` sections.

The compiler keyword `__no_init` places variables into sections that must be handled by a `do not initialize` directive.

Example

```
/* Do not initialize read-write sections whose name ends with
   __noinit at program start */
do not initialize { rw section .*_noinit };
place in RAM { rw section .*_noinit };
```

See also *Initialization at system startup, page 88, and Initialize directive, page 392.*

Keep directive

Syntax

```
keep
{
    section-selectors
}
[except
{
    section-selectors
}];
```

For information about extended selectors and except clauses, see *Section selection, page 396*.

Description

The `keep` directive specifies that all selected sections should be kept in the executable image, even if there are no references to the sections.

Example

```
keep { section .keep* } except {section .keep};
```

Place at directive

Syntax

```
[ "name": ]
place at { address [ memory: ] expr | start of region_expr |
          end of region_expr }
{
    extended-selectors
}
[except
{
    section-selectors
}];
```

For information about extended selectors and except clauses, see *Section selection*, page 396.

Parameters

<i>memory: expr</i>	A specific address in a specific memory. The address must be available in the supplied memory defined by the <code>define memory</code> directive. The memory specifier is optional if there is only one memory.
<i>start of region_expr</i>	A region expression. The start of the region is used.
<i>end of region_expr</i>	A region expression. The end of the region is used.

Description

The `place at` directive places sections and blocks either at a specific address or, at the beginning or the end of a region. The same address cannot be used for two different `place at` directives. It is also not possible to use an empty region in a `place at` directive. If placed in a region, the sections and blocks will be placed before any other sections or blocks placed in the same region with a `place in` directive.

The sections and blocks will be placed in the region in an arbitrary order. To specify a specific order, use the `block` directive.

The *name*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only section .startup at the beginning of the
   code_region */
"START": place at start of ROM { readonly section .startup };
```

See also

Place in directive, page 396.

Place in directive

Syntax

```
[ "name": ]  
place in region-expr  
{  
    extended-selectors  
}  
[except{  
    section-selectors  
}];
```

where *region-expr* is a region expression, see also *Regions*, page 385.

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 396.

Description

The `place in` directive places sections and blocks in a specific region. The sections and blocks will be placed in the region in an arbitrary order.

To specify a specific order, use the `block` directive. The region can have several ranges.

The *name*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only sections in the code_region */  
"ROM": place in ROM { readonly };
```

See also

Place at directive, page 395.

Section selection

The purpose of *section selection* is to specify—by means of *section selectors* and *except clauses*—the sections that an `ILINK` directive should be applied to. All sections that match one or more of the section selectors will be selected, and none of the sections selectors in the `except` clause, if any. Each section selector can match sections on section attributes, section name, and object or library name.

Some directives provide functionality that requires more detailed selection capabilities, for example directives that can be applied on both sections and blocks. In this case, the *extended-selectors* are used.

Section-selectors

Syntax

```
{ [ section-selector ] [, section-selector... ] }  
where section-selector is:  
    [ section-attribute ] [ section-type ] [ section sectionname ]  
    [object {module | filename} ]  
where section-attribute is:  
    [ ro [ code | data ] | rw [ code | data ] | zi ]  
and where ro, rw, and zi also can be readonly, readwrite, and zeroinit,  
respectively.  
And section-type is:  
    [ preinit_array | init_array ]
```

Parameters

ro or readonly	Read-only sections.
rw or readwrite	Read/write sections.
zi or zeroinit	Zero-initialized sections. These sections have no content and should possibly be initialized with zeros during system startup.
code	Sections that contain code.
data	Sections that contain data.
preinit_array	Sections of the ELF section type SHT_PREINIT_ARRAY.
init_array	Sections of the ELF section type SHT_INIT_ARRAY.
sectionname	The section name. Two wildcards are allowed: ? matches any single character * matches zero or more characters.
module	A name in the form <i>objectname(libraryname)</i> . Sections from object modules where both the object name and the library name match their respective patterns are selected. An empty library name pattern selects only sections from object files.
filename	The name of an object file, a library, or an object in a library. Two wildcards are allowed: ? matches any single character * matches zero or more characters.

Description

A section selector selects all sections that match the section attribute, section type, section name, and the name of the *object*, where *object* is an object file, a library, or an object in a library. Up to three of the four conditions can be omitted. If the section attribute is omitted, any section will be selected, without restrictions on the section attribute. If the section type is omitted, sections of any type will be selected.

If the section name part or the object name part is omitted, sections will be selected without restrictions on the section name or object name, respectively.

It is also possible to use only { } without any section selectors, which can be useful when defining blocks.

Note that a section selector with narrower scope has higher priority than a more generic section selector.

If more than one section selector matches for the same purpose, one of them must be more specific. A section selector is more specific than another one if:

- It specifies a section type and the other one does not
- It specifies a section name or object name with no wildcards and the other one does not
- There could be sections that match the other selector that also match this one, and the reverse is not true.

Selector 1	Selector 2	More specific
section "foo*"	section "f*"	Selector 1
section "*x"	section "f*"	Neither
ro code section "f*"	ro section "f*"	Selector 1
init_array	ro section "xx"	Selector 1
section ".intvec"	ro section ".int*"	Selector 1
section ".intvec"	object "foo.o"	Neither

Table 41: Examples of section selector specifications

Example

```
{ rw } /* Selects all read-write sections */

{ section .mydata* } /* Selects only .mydata* sections */
/* Selects .mydata* sections available in the object special.o */
{ section .mydata* object special.o }
```

Assuming a section in an object named `foo.o` in a library named `lib.a`, any of these selectors will select that section:

```
object foo.o(lib.a)
object f*(lib*)
object foo.o
```

object lib.a

See also

Initialize directive, page 392, Do not initialize directive, page 394, and Keep directive, page 394.

Extended-selectors

Syntax	<pre>{ [<i>extended-selector</i>] [, <i>extended-selector</i>...] }</pre> <p>where <i>extended-selector</i> is:</p> <pre>[first last] { <i>section-selector</i> block name [<i>inline-block-def</i>] overlay name }</pre> <p>where <i>inline-block-def</i> is:</p> <pre>[block-params] <i>extended-selectors</i></pre>									
Parameters	<table><tr><td>first</td><td>Places the selected name first in the region, block, or overlay.</td></tr><tr><td>last</td><td>Places the selected name last in the region, block, or overlay.</td></tr><tr><td>block</td><td>The name of the block.</td></tr><tr><td>overlay</td><td>The name of the overlay.</td></tr></table>		first	Places the selected name first in the region, block, or overlay.	last	Places the selected name last in the region, block, or overlay.	block	The name of the block.	overlay	The name of the overlay.
first	Places the selected name first in the region, block, or overlay.									
last	Places the selected name last in the region, block, or overlay.									
block	The name of the block.									
overlay	The name of the overlay.									
Description	In addition to what the <i>section-selector</i> does, <i>extended-selector</i> provides functionality for placing blocks or overlays first or last in a set of sections, a block, or an overlay. It is also possible to create an <i>inline</i> definition of a block. This means that you can get more precise control over section placement.									
Example	<pre>define block First { section .first }; /* Define a block holding the section .first */ define block Table { first block First }; /* Define a block where the block First is placed first */</pre> <p>or, equivalently using an inline definition of the block First:</p> <pre>define block Table { first block First { section .first } };</pre>									
See also	<i>Define block directive</i> , page 389, <i>Define overlay directive</i> , page 390, and <i>Place at directive</i> , page 395.									

Using symbols, expressions, and numbers

In the linker configuration file, you can also:

- Define and export symbols
The `define symbol` directive defines a symbol with a specified value that can be used in expressions in the configuration file. The symbol can also be exported to be used by the application or the debugger. See *Define symbol directive, page 400*, and *Export directive, page 401*.
- Use expressions and numbers
In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, etc. See *Expressions, page 401*.

Define symbol directive

Syntax	<code>define [exported] symbol <i>name</i> = <i>expr</i>;</code>	
Parameters	<code>exported</code>	Exports the symbol to be usable by the executable image.
	<code><i>name</i></code>	The name of the symbol.
	<code><i>expr</i></code>	The symbol value.
Description	<p>The <code>define symbol</code> directive defines a symbol with a specified value. The symbol can then be used in expressions in the configuration file. The symbols defined in this way work exactly like the symbols defined with the option <code>--config_def</code> outside of the configuration file.</p> <p>The <code>define exported symbol</code> variant of this directive is a shortcut for using the directive <code>define symbol</code> in combination with the <code>export symbol</code> directive. On the command line this would require both a <code>--config_def</code> option and a <code>--define_symbol</code> option to achieve the same effect.</p>	
	<p>Note:</p> <ul style="list-style-type: none">● A symbol cannot be redefined● Symbols that are either prefixed by <code>__X</code>, where <code>X</code> is a capital letter, or that contain <code>__</code> (double underscore) are reserved for toolset vendors.	
Example	<pre>/* Define the symbol my_symbol with the value 4 */ define symbol my_symbol = 4;</pre>	

See also *Export directive, page 401* and *Interaction between ILINK and the application, page 103*.

Export directive

Syntax	<code>export symbol name;</code>	
Parameters	<i>name</i>	The name of the symbol.
Description	The <code>export</code> directive defines a symbol to be exported, so that it can be used both from the executable image and from a global label. The application, or the debugger, can then refer to it for setup purposes etc.	
Example	<pre>/* Define the symbol my_symbol to be exported */ export symbol my_symbol;</pre>	

Expressions

Syntax

An expression is built up of the following constituents:

```
expression binop expression
unop expression
expression ? expression : expression
(expression)
number
symbol
func-operator
```

where *binop* is one of these binary operators:

```
+, -, *, /, %, <<, >>, <, >, ==, !=, &, ^, |, &&, ||
```

where *unop* is one of this unary operators:

```
+, -, !, ~
```

where *number* is a number, see *Numbers, page 402*

where *symbol* is a defined symbol, see *Define symbol directive, page 400* and *--config_def, page 279*

and where *func-operator* is one of these function-like operators:

<code>minimum(<i>expr</i>, <i>expr</i>)</code>	Returns the smallest of the two parameters.
--	---

<code>maximum(<i>expr</i>, <i>expr</i>)</code>	Returns the largest of the two parameters.
<code>isempty(<i>r</i>)</code>	Returns True if the region is empty, otherwise False.
<code>isdefinedsymbol(<i>expr-symbol</i>)</code>	Returns True if the expression symbol is defined, otherwise False.
<code>start(<i>r</i>)</code>	Returns the lowest address in the region.
<code>end(<i>r</i>)</code>	Returns the highest address in the region.
<code>size(<i>r</i>)</code>	Returns the size of the complete region.

where *expr* is an expression, and *r* is a region expression, see *Region expression*, page 387.

Description

In the linker configuration file, an expression is a 65-bit value with the range -2⁶⁴ to 2⁶⁴. The expression syntax closely follows C syntax with some minor exceptions. There are no assignments, casts, pre- or post-operations, and no address operations (*, &, [], ->, and .). Some operations that extract a value from a region expression, etc, use a syntax resembling that of a function call. A boolean expression returns 0 (False) or 1 (True).

Numbers

Syntax

`nr [nr-suffix]`
where *nr* is either a decimal number or a hexadecimal number (0x... or 0X...).
and where *nr-suffix* is one of:

```
K      /* Kilo = (1 << 10) 1024 */
M      /* Mega = (1 << 20) 1048576 */
G      /* Giga = (1 << 30) 1073741824 */
T      /* Tera = (1 << 40) 1099511627776 */
P      /* Peta = (1 << 50) 1125899906842624 */
```

Description

A number can be expressed either by normal C means or by suffixing it with a set of useful suffixes, which provides a compact way of specifying numbers.

Example

1024 is the same as 0x400, which is the same as 1K.

Structural configuration

The structural directives provide means for creating structure within the configuration, such as:

- Conditional inclusion
An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations in the same file. See *If directive*, page 403.
- Dividing the linker configuration file into several different files
The `include` directive makes it possible to divide the configuration file into several logically distinct files. See *Include directive*, page 403.

If directive

Syntax

```
if (expr) {  
    directives  
[ } else if (expr) {  
    directives ]  
[ } else {  
    directives ]  
}
```

where *expr* is an expression, see *Expressions*, page 401.

Parameters

directives

Any ILINK directive.

Description

An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations, for example both a banked and non-banked memory configuration, in the same file.

The directives inside an `if` part, `else if` part, or an `else` part are syntax checked and processed regardless of whether the conditional expression was true or false, but only the directives in the part where the conditional expression was true, or the `else` part if none of the conditions were true, will have any effect outside the `if` directive. The `if` directives can be nested.

Example

See *Empty region*, page 388.

Include directive

Syntax	<code>include filename;</code>	
Parameters	<i>filename</i>	A string literal where both / and \ can be used as the directory delimiter.
Description	The <code>include</code> directive makes it possible to divide the configuration file into several logically distinct parts, each in a separate file. For instance, there might be parts that you need to change often and parts that you seldom edit.	

Section reference

The compiler places code and data into sections. Based on a configuration specified in the linker configuration file, ILINK places sections in memory.

This chapter lists all predefined ELF sections and blocks that are available for the IAR build tools for ARM, and gives detailed reference information about each section.

For more information about sections, see the chapter *Modules and sections*, page 82.

Summary of sections

This table lists the ELF sections and blocks that are used by the IAR build tools:

Section	Description
.bss	Holds zero-initialized static and global variables.
CSTACK	Holds the stack used by C or C++ programs.
.cstart	Holds the startup code.
.data	Holds static and global initialized variables.
.data_init	Holds initial values for .data sections when the linker directive <code>initialize by copy</code> is used.
__DLIB_PERTHREAD	Holds variables that contain static states for DLIB modules.
.exc.text	Holds exception-related code.
HEAP	Holds the heap used for dynamically allocated data.
.iar.dynexit	Holds the <code>atexit</code> table.
.init_array	Holds a table of dynamic initialization functions.
.intvec	Holds the reset vector table
IRQ_STACK	Holds the stack for interrupt requests, IRQ, and exceptions.
.noinit	Holds <code>__no_init</code> static and global variables.
.preinit_array	Holds a table of dynamic initialization functions.
.prepreinit_array	Holds a table of dynamic initialization functions.
.rodata	Holds constant data.

Table 42: Section summary

Section	Description
.text	Holds the program code.
.textrw	Holds __ramfunc declared program code.
.textrw_init	Holds initializers for the .textrw declared section.

Table 42: Section summary (Continued)

In addition to the ELF sections used for your application, the tools use a number of other ELF sections for a variety of purposes:

- Sections starting with .debug generally contain debug information in the DWARF format
- Sections starting with .iar.debug contain supplemental debug information in an IAR format
- The section .comment contains the tools and command lines used for building the file
- Sections starting with .rel or .rela contain ELF relocation information
- The section .symtab contains the symbol table for a file
- The section .strtab contains the names of the symbol in the symbol table
- The section .shstrtab contains the names of the sections.

Descriptions of sections and blocks

This section gives reference information about each section, where the:

- *Description* describes what type of content the section is holding and, where required, how the section is treated by the linker
- *Memory placement* describes memory placement restrictions.

For information about how to allocate sections in memory by modifying the linker configuration file, see *Placing code and data—the linker configuration file*, page 85.

.bss

Description	Holds zero-initialized static and global variables.
Memory placement	This section can be placed anywhere in memory.

CSTACK

Description	Block that holds the internal data stack.
Memory placement	This block can be placed anywhere in memory.
See also	<i>Setting up the stack, page 99.</i>

.cstart

Description	Holds the startup code.
Memory placement	This section can be placed anywhere in memory.

.data

Description	Holds static and global initialized variables. In object files, this includes the initial values. When the linker directive <code>initialize by copy</code> is used, a corresponding <code>.data_init</code> section is created for each <code>.data</code> section, holding the possibly compressed initial values.
Memory placement	This section can be placed anywhere in memory.

.data_init

Description	Holds the possibly compressed initial values for <code>.data</code> sections. This section is created by the linker if the <code>initialize by copy</code> linker directive is used.
Memory placement	This section can be placed anywhere in memory.

__DLIB_PERTHREAD

Description	Holds variables that contain static states for DLIB modules.
Memory placement	This section is automatically placed correctly; you must not change its placement.
See also	<i>Managing a multithreaded environment, page 136.</i>

.exc.text

Description	Holds code that is only executed when your application handles an exception.
Memory placement	In the same memory as <code>.text</code> .
See also	<i>Exception handling, page 178.</i>

HEAP

Description	Holds the heap used for dynamically allocated data in memory, in other words data allocated by <code>malloc</code> and <code>free</code> , and in C++, <code>new</code> and <code>delete</code> .
Memory placement	This section can be placed anywhere in memory.
See also	<i>Setting up the heap, page 100.</i>

.iar.dynexit

Description	Holds the table of calls to be made at exit.
Memory placement	This section can be placed anywhere in memory.
See also	<i>Setting up the atexit limit, page 100.</i>

.init_array

Description	Holds pointers to routines to call for initializing one or more C++ objects with static storage duration.
Memory placement	This section can be placed anywhere in memory.

.intvec

Description	Holds the reset vector and exception vectors which contain branch instructions to <code>cstartup</code> , interrupt service routines etc.
Memory placement	Must be placed at address range <code>0x00</code> to <code>0x3F</code> .

IRQ_STACK

Description	Holds the stack which is used when servicing IRQ exceptions. Other stacks may be added as needed for servicing other exception types: FIQ, SVC, ABT, and UND. The <code>cstartup.s</code> file must be modified to initialize the exception stack pointers used.
	Note: This section is not used when compiling for Cortex-M.
Memory placement	This section can be placed anywhere in memory.
See also	<i>Exception stacks, page 188.</i>

.noinit

Description	Holds static and global <code>__no_init</code> variables.
Memory placement	This section can be placed anywhere in memory.

.preinit_array

Description	Like <code>.init_array</code> , but is used by the library to make some C++ initializations happen before the others.
Memory placement	This section can be placed anywhere in memory.
See also	<i>.init_array, page 408.</i>

.prepreinit_array

Description	Like <code>.init_array</code> , but is used when C static initialization is rewritten as dynamic initialization. Performed before all C++ dynamic initialization.
Memory placement	This section can be placed anywhere in memory.
See also	<i>.init_array, page 408.</i>

.rodata

Description	Holds constant data. This can include constant variables, string and aggregate literals, etc.
Memory placement	This section can be placed anywhere in memory.

.text

Description	Holds program code, except the code for system initialization.
Memory placement	This section can be placed anywhere in memory.

.textrw

Description	Holds <code>__ramfunc</code> declared program code.
Memory placement	This section can be placed anywhere in memory.
See also	<code>__ramfunc</code> , page 311.

.textrw_init

Description	Holds initializers for the <code>.textrw</code> declared sections.
Memory placement	This section can be placed anywhere in memory.
See also	<code>__ramfunc</code> , page 311.

Stack usage control files

This chapter describes the purpose and the syntax of stack usage control files.

Before you read this chapter, see *Stack usage analysis*, page 90.

Overview

A stack usage control file consists of a sequence of directives that control stack usage analysis. You can use C ("*/*...*/*") and C++ ("*//...*") comments in these files.

The default filename extension for stack usage control files is *suc*.

Stack usage control directives

This section gives detailed reference information about each stack usage control directive.

function directive

Syntax

```
[ override ] function [ category ] function-spec : stack-size  
[ , call-info... ];
```

category See *category*, page 413

function-spec See *function-spec*, page 413

call-info See *call-info*, page 414

stack-size See *stack-size*, page 415

Description

Specifies what the maximum stack usage is in a function and which other functions that are called from that function.

Normally, an error is issued if there already is stack usage information for the function, but if you start with *override*, the error will be suppressed and the information supplied in the directive will be used instead of the previous information.

Examples	<pre>function MyFunc1: 32, calls MyFunc2, calls MyFunc3, MyFunc4: 16; function [interrupt] nmi: 44</pre>
----------	---

exclude directive

Syntax	<code>exclude <i>function-spec</i> [, <i>function-spec</i>...];</code>		
Parameters	<table><tr><td><i>function-spec</i></td><td>See <i>function-spec</i>, page 413</td></tr></table>	<i>function-spec</i>	See <i>function-spec</i> , page 413
<i>function-spec</i>	See <i>function-spec</i> , page 413		
Description	Excludes the specified functions, and call trees originating with them, from stack usage calculations.		
Example	<code>exclude fun1, fun2;</code>		

possible calls directive

Syntax	<code>possible calls <i>calling-func</i> : <i>called-func</i> [, <i>called-func</i>...];</code>				
Parameters	<table><tr><td><i>calling-func</i></td><td>See <i>function-spec</i>, page 413</td></tr><tr><td><i>called-func</i></td><td>See <i>function-spec</i>, page 413</td></tr></table>	<i>calling-func</i>	See <i>function-spec</i> , page 413	<i>called-func</i>	See <i>function-spec</i> , page 413
<i>calling-func</i>	See <i>function-spec</i> , page 413				
<i>called-func</i>	See <i>function-spec</i> , page 413				
Description	Specifies an exhaustive list of possible destinations for all indirect calls in one function. Use this for functions which are known to perform indirect calls and where you know exactly which functions that might be called in this particular application. Consider using the <code>#pragma calls</code> directive if the information about which functions that might be called is available when compiling.				
Example	<code>possible calls afun: bfun, cfun;</code>				
See also	<i>calls</i> , page 320.				

call graph root directive

Syntax	<pre>call graph root [category] : function-spec [, function-spec...];</pre>	
Parameters	<i>category</i>	See <i>category</i> , page 413
	<i>function-spec</i>	See <i>function-spec</i> , page 413
Description	<p>Specifies that the listed functions are call graph roots. You can optionally specify a call graph root category. Call graph roots are listed under their category in the <i>Stack Usage</i> chapter in the linker map file.</p> <p>The linker will normally issue a warning for functions needed in the application that are not call graph roots and which do not appear to be called.</p>	
Example	<pre>call graph root [task]: fun1, fun2;</pre>	
See also	<i>call_graph_root</i> , page 320.	

Syntactic components

The stack usage control directives use some syntactical components. These are described below.

category

Syntax	<pre>[name]</pre>
Description	A call graph root category. You can use any name you like.
Examples	<pre>[interrupt]</pre>
	<pre>[task]</pre>

function-spec

Syntax	<pre>name [[module-spec]]</pre>
Description	The name of a function, and for module-local functions, the name of the module it is defined in.

Example	<code>xFun</code> <code>MyFun [file.o]</code>
---------	--

module-spec

Syntax	<code>name [(name)]</code>
Description	<p>Specifies the name of a module, and optionally, in parentheses, the name of the library it belongs to. To distinguish between modules with the same name, you can specify:</p> <ul style="list-style-type: none">• The complete path of the file ("D:\C1\test\file.o")• As many path elements as are needed at the end of the path ("test\file.o")• Some path elements at the start of the path, followed by "...", followed by some path elements at the end ("D:\...\file.o").

Example	<code>file.o</code> <code>file.o(lib.a)</code> <code>"D:\C1\test\file.o"</code>
---------	---

name

Description	<p>A name can be either an identifier or a quoted string.</p> <p>The first character of an identifier must be either a letter or one of the characters "_", "\$", or ".". The rest of the characters can also be digits.</p> <p>A quoted string starts and ends with " and can contain any character. Two consecutive " characters can be used inside a quoted string to represent a single ".</p>
-------------	--

Examples	<code>MyFun</code> <code>file.o</code> <code>"file-1.o"</code>
----------	--

call-info

Syntax	<code>calls function-spec [, function-spec...] [: stack-size]</code>
Description	<p>Specifies one or more called functions, and optionally, the stack size at the calls.</p>

Examples	<code>calls MyFunc1 : stack 16</code> <code>calls MyFunc2, MyFunc3, MyFunc4</code>
----------	---

stack-size

Syntax	<code>[stack] <i>size</i></code>
Description	Specifies the size of a stack frame.
Examples	<code>24</code> <code>stack 28</code>

IAR utilities

This chapter describes the IAR command line utilities that are available:

- The IAR Archive Tool—`iarchive`—creates and manipulates a library (an archive) of several ELF object files
- The IAR ELF Tool—`ielftool`—performs various transformations on an ELF executable image (such as fill, checksum, format conversions, etc)
- The IAR ELF Dumper for ARM—`ielfdumparm`—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—`iobjmanip`—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—`isymexport`—exports absolute symbols from a ROM image file, so that they can be used when you link an add-on application.

Descriptions of options gives detailed reference information about each command line option available for the different utilities.

The IAR Archive Tool—`iarchive`

The IAR Archive Tool, `iarchive`, can create a library (an archive) file from several ELF object files. You can also use `iarchive` to manipulate ELF libraries.

A library file contains several relocatable ELF object modules, each of which can be independently used by a linker. In contrast with object modules specified directly to the linker, each module in a library is only included if it is needed.

For information about how to build a library in the IDE, see the *IDE Project Management and Building Guide for ARM®*.

INVOCATION SYNTAX

The invocation syntax for the archive builder is:

```
iarchive parameters
```

Parameters

The parameters are:

Parameter	Description
<i>command</i>	Command line options that define an operation to be performed. Such an option must be specified before the name of the library file.
<i>libraryfile</i>	The library file to be operated on.
<i>objectfile1 ... objectfileN</i>	The object file(s) that the specified command operates on.
<i>options</i>	Command line options that define actions to be performed. These options can be placed anywhere on the command line.

Table 43: iarchive parameters

Examples

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

```
iarchive mylibrary.a module1.o module2.o module3.o.
```

This example lists the contents of `mylibrary.a`:

```
iarchive --toc mylibrary.a
```

This example replaces `module3.o` in the library with the content in the `module3.o` file and appends `module4.o` to `mylibrary.a`:

```
iarchive --replace mylibrary.a module3.o module4.o
```

SUMMARY OF IARCHIVE COMMANDS

This table summarizes the `iarchive` commands:

Command line option	Description
<code>--create</code>	Creates a library that contains the listed object files.
<code>--delete, -d</code>	Deletes the listed object files from the library.
<code>--extract, -x</code>	Extracts the listed object files from the library.
<code>--replace, -r</code>	Replaces or appends the listed object files to the library.
<code>--symbols</code>	Lists all symbols defined by files in the library.
<code>--toc, -t</code>	Lists all files in the library.

Table 44: iarchive commands summary

For more information, see *Descriptions of options, page 431*.

SUMMARY OF IARCHIVE OPTIONS

This table summarizes the `iarchive` options:

Command line option	Description
<code>-f</code>	Extends the command line.
<code>--output, -o</code>	Specifies the library file.
<code>--silent</code>	Sets silent operation.
<code>--verbose, -V</code>	Reports all performed operations.

Table 45: *iarchive* options summary

For more information, see *Descriptions of options*, page 431.

DIAGNOSTIC MESSAGES

This section lists the messages produced by `iarchive`:

La001: could not open file *filename*

`iarchive` failed to open an object file.

La002: illegal path *pathname*

The path *pathname* is not a valid path.

La006: too many parameters to *cmd* command

A list of object modules was specified as parameters to a command that only accepts a single library file.

La007: too few parameters to *cmd* command

A command that takes a list of object modules was issued without the expected modules.

La008: *lib* is not a library file

The library file did not pass a basic syntax check. Most likely the file is not the intended library file.

La009: *lib* has no symbol table

The library file does not contain the expected symbol information. The reason might be that the file is not the intended library file, or that it does not contain any ELF object modules.

La010: no library parameter given

The tool could not identify which library file to operate on. The reason might be that a library file has not been specified.

La011: file *file* already exists

The file could not be created because a file with the same name already exists.

La013: file confusions, *lib* given as both library and object

The library file was also mentioned in the list of object modules.

La014: module *module* not present in archive *lib*

The specified object module could not be found in the archive.

La015: internal error

The invocation triggered an unexpected error in `iarchive`.

Ms003: could not open file *filename* for writing

`iarchive` failed to open the archive file for writing. Make sure that it is not write protected.

Ms004: problem writing to file *filename*

An error occurred while writing to file *filename*. A possible reason for this is that the volume is full.

Ms005: problem closing file *filename*

An error occurred while closing the file *filename*.

The IAR ELF Tool—ielftool

The IAR ELF Tool, `ielftool`, can generate a checksum on specific ranges of memories. This checksum can be compared with a checksum calculated on your application.

The source code for `ielftool` and a Microsoft VisualStudio 2005 template project are available in the `arm\src\elfutils` directory. If you have specific requirements for how the checksum should be generated or requirements for format conversion, you can modify the source code accordingly.

INVOCATION SYNTAX

The invocation syntax for the IAR ELF Tool is:

```
ielftool [options] inputfile outputfile [options]
```

The `ielftool` tool will first process all the fill options, then it will process all the checksum options (from left to right).

Parameters

The parameters are:

Parameter	Description
<i>inputfile</i>	An absolute ELF executable image produced by the ILINK linker.
<i>options</i>	Any of the available command line options, see <i>Summary of ielftool options, page 421</i> .
<i>outputfile</i>	An absolute ELF executable image.

Table 46: ielftool parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters, page 234*.

Example

This example fills a memory range with 0xFF and then calculates a checksum on the same range:

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
--checksum __checksum:4,crc32;0-0xFF
```

SUMMARY OF IELFTOOL OPTIONS

This table summarizes the `ielftool` command line options:

Command line option	Description
<code>--bin</code>	Sets the format of the output file to binary.
<code>--checksum</code>	Generates a checksum.
<code>--fill</code>	Specifies fill requirements.
<code>--ihex</code>	Sets the format of the output file to linear Intel hex.
<code>--self_reloc</code>	Not for general use.
<code>--silent</code>	Sets silent operation.
<code>--simple</code>	Sets the format of the output file to Simple code.
<code>--srec</code>	Sets the format of the output file to Motorola S-records.

Table 47: ielftool options summary

Command line option	Description
--srec-len	Restricts the number of data bytes in each S-record.
--srec-s3only	Restricts the S-record output to contain only a subset of records.
--strip	Removes debug information.
--verbose, -V	Prints all performed operations.

Table 47: ielftool options summary (Continued)

For more information, see *Descriptions of options*, page 431.

The IAR ELF Dumper for ARM—ielfdumparm

The IAR ELF Dumper for ARM, `ielfdumparm`, can be used for creating a text representation of the contents of a relocatable or absolute ELF file.

`ielfdumparm` can be used in one of three ways:

- To produce a listing of the general properties of the input file and the ELF segments and ELF sections it contains. This is the default behavior when no command line options are used.
- To also include a textual representation of the contents of each ELF section in the input file. To specify this behavior, use the command line option `--all`.
- To produce a textual representation of selected ELF sections from the input file. To specify this behavior, use the command line option `--section`.

INVOCATION SYNTAX

The invocation syntax for `ielfdumparm` is:

```
ielfdumparm input_file [output_file]
```

Note: `ielfdumparm` is a command line tool which is not primarily intended to be used in the IDE.

Parameters

The parameters are:

Parameter	Description
<code>input_file</code>	An ELF relocatable or executable file to use as input.
<code>output_file</code>	A file or directory where the output is emitted. If absent and no <code>--output</code> option is specified, output is directed to the console.

Table 48: ielfdumparm parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 234.

SUMMARY OF IELFDUMPARM OPTIONS

This table summarizes the `ielfdumparm` command line options:

Command line option	Description
<code>--all</code>	Generates output for all input sections regardless of their names or numbers.
<code>--code</code>	Dumps all sections that contain executable code.
<code>-f</code>	Extends the command line.
<code>--output, -o</code>	Specifies an output file.
<code>--no_strtab</code>	Suppresses dumping of string table sections.
<code>--raw</code>	Uses the generic hexadecimal/ASCII output format for the contents of any selected section, instead of any dedicated output format for that section.
<code>--section, -s</code>	Generates output for selected input sections.

Table 49: `ielfdumparm` options summary

For more information, see *Descriptions of options*, page 431.

The IAR ELF Object Tool—`iobjmanip`

Use the IAR ELF Object Tool, `iobjmanip`, to perform low-level manipulation of ELF object files.

INVOCATION SYNTAX

The invocation syntax for the IAR ELF Object Tool is:

`iobjmanip options inputfile outputfile`

Parameters

The parameters are:

Parameter	Description
<code>options</code>	Command line options that define actions to be performed. These options can be placed anywhere on the command line. At least one of the options must be specified.
<code>inputfile</code>	A relocatable ELF object file.

Table 50: `iobjmanip` parameters

Parameter	Description
<i>outputfile</i>	A relocatable ELF object file with all the requested operations applied.

Table 50: iobjmanip parameters (Continued)

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 234.

Examples

This example renames the section `.example` in `input.o` to `.example2` and stores the result in `output.o`:

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

SUMMARY OF IOBJMANIP OPTIONS

This table summarizes the `iobjmanip` options:

Command line option	Description
<code>-f</code>	Extends the command line.
<code>--remove_section</code>	Removes a section.
<code>--rename_section</code>	Renames a section.
<code>--rename_symbol</code>	Renames a symbol.
<code>--strip</code>	Removes debug information.

Table 51: iobjmanip options summary

For more information, see *Descriptions of options*, page 431.

DIAGNOSTIC MESSAGES

This section lists the messages produced by `iobjmanip`:

Lm001: No operation given

None of the command line parameters specified an operation to perform.

Lm002: Expected *nr* parameters but got *nr*

Too few or too many parameters. Check invocation syntax for `iobjmanip` and for the used command line options.

Lm003: Invalid section/symbol renaming pattern *pattern*

The pattern does not define a valid renaming operation.

Lm004: Could not open file *filename*

`iobjmanip` failed to open the input file.

Lm005: ELF format error *msg*

The input file is not a valid ELF object file.

Lm006: Unsupported section type *nr*

The object file contains a section that `iobjmanip` cannot handle. This section will be ignored when generating the output file.

Lm007: Unknown section type *nr*

`iobjmanip` encountered an unrecognized section. `iobjmanip` will try to copy the content as is.

Lm008: Symbol *symbol* has unsupported format

`iobjmanip` encountered a symbol that cannot be handled. `iobjmanip` will ignore this symbol when generating the output file.

Lm009: Group type *nr* not supported

`iobjmanip` only supports groups of type `GRP_COMDAT`. If any other group type is encountered, the result is undefined.

Lm010: Unsupported ELF feature in *file*: *msg*

The input file uses a feature that `iobjmanip` does not support.

Lm011: Unsupported ELF file type

The input file is not a relocatable object file.

Lm012: Ambiguous rename for section/symbol *name* (*alt1* and *alt2*)

An ambiguity was detected while renaming a section or symbol. One of the alternatives will be used.

Lm013: Section *name* removed due to transitive dependency on *name*

A section was removed as it depends on an explicitly removed section.

Lm014: File has no section with index *nr*

A section index, used as a parameter to `--remove_section` or `--rename_section`, did not refer to a section in the input file.

Ms003: could not open file *filename* for writing

`iobjmanip` failed to open the output file for writing. Make sure that it is not write protected.

Ms004: problem writing to file *filename*

An error occurred while writing to file *filename*. A possible reason for this is that the volume is full.

Ms005: problem closing file *filename*

An error occurred while closing the file *filename*.

The IAR Absolute Symbol Exporter—ismexport

The IAR Absolute Symbol Exporter, `ismexport`, can export absolute symbols from a ROM image file, so that they can be used when you link an add-on application.

INVOCATION SYNTAX

The invocation syntax for the IAR Absolute Symbol Exporter is:

`ismexport [options] inputfile outputfile [options]`

Parameters

The parameters are:

Parameter	Description
<i>inputfile</i>	A ROM image in the form of an executable ELF file (output from linking).
<i>options</i>	Any of the available command line options, see <i>Summary of ismexport options</i> , page 427.
<i>outputfile</i>	A relocatable ELF file that can be used as input to linking, and which contains all or a selection of the absolute symbols in the input file. The output file contains only the symbols, not the actual code or data sections. A steering file can be used to control which symbols that are included, and also to rename some of the symbols if that is desired.

Table 52: *ielftool* parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 234.

SUMMARY OF ISYMEXPORT OPTIONS

This table summarizes the `isymexport` command line options:

Command line option	Description
<code>--edit</code>	Specifies a steering file.
<code>-f</code>	Extends the command line.
<code>--ram_reserve_ranges</code>	Generates symbols to reserve the areas in RAM that the image uses.
<code>--reserve_ranges</code>	Generates symbols to reserve the areas in ROM and RAM that the image uses.

Table 53: *isymexport* options summary

For more information, see *Descriptions of options*, page 431.

STEERING FILES

A steering file can be used for controlling which symbols that are included, and also to rename some of the symbols if that is desired. In the file, you can use `show` and `hide` directives to select which public symbols from the input file that are to be included in the output file. `rename` directives can be used for changing the names of symbols in the input file.

Syntax

The following syntax rules apply:

- Each directive is specified on a separate line.
- C comments (`/*...*/`) and C++ comments (`//...`) can be used.
- Patterns can contain wildcard characters that match more than one possible character in a symbol name.
- The `*` character matches any sequence of zero or more characters in a symbol name.
- The `?` character matches any single character in a symbol name.

Example

```
rename xxx_* as YY_* /*Change symbol prefix from xxx_ to YY_* */
show YY_*           /* Export all symbols from YY package */
hide *_internal     /* But do not export internal symbols */
show zzz?           /* Export zzza, but not zzzaaa */
hide zzzx           /* But do not export zzzx */
```

Show directive

Syntax	<code>show pattern</code>	
Parameters	<code>pattern</code>	A pattern to match against a symbol name.
Description	A symbol with a name that matches the pattern will be included in the output file unless this is overridden by a later <code>hide</code> directive.	
Example	<pre>/* Include all public symbols ending in _pub. */ show *_pub</pre>	

Hide directive

Syntax	<code>hide pattern</code>	
Parameters	<code>pattern</code>	A pattern to match against a symbol name.
Description	A symbol with a name that matches the pattern will not be included in the output file unless this is overridden by a later <code>show</code> directive.	
Example	<pre>/* Do not include public symbols ending in _sys. */ hide *_sys</pre>	

Rename directive

Syntax	<code>rename pattern1 pattern2</code>	
Parameters	<code>pattern1</code>	A pattern used for finding symbols to be renamed. The pattern can contain no more than one <code>*</code> or <code>?</code> wildcard character.
	<code>pattern2</code>	A pattern used for the new name for a symbol. If the pattern contains a wildcard character, it must be of the same kind as in <code>pattern1</code> .
Description	Use this directive to rename symbols from the output file to the input file. No exported symbol is allowed to match more than one <code>rename</code> pattern.	

`rename` directives can be placed anywhere in the steering file, but they are executed before any `show` and `hide` directives. Thus, if a symbol will be renamed, all `show` and `hide` directives in the steering file must refer to the new name.

If the name of a symbol matches a *pattern1* pattern that contains no wildcard characters, the symbol will be renamed *pattern2* in the output file.

If the name of a symbol matches a *pattern1* pattern that contains a wildcard character, the symbol will be renamed *pattern2* in the output file, with part of the name matching the wildcard character preserved.

Example

```
/* xxx_start will be renamed Y_start_X in the output file,
   xxx_stop will be renamed Y_stop_X in the output file. */
rename xxx_* Y_*_X
```

DIAGNOSTIC MESSAGES

This section lists the messages produced by `ismexport`:

Es001: could not open file *filename*

`ismexport` failed to open the specified file.

Es002: illegal path *pathname*

The path *pathname* is not a valid path.

Es003: format error: *message*

A problem occurred while reading the input file.

Es004: no input file

No input file was specified.

Es005: no output file

An input file, but no output file was specified.

Es006: too many input files

More than two files were specified.

Es007: input file is not an ELF executable

The input file is not an ELF executable file.

Es008: unknown directive: *directive*

The specified directive in the steering file is not recognized.

Es009: unexpected end of file

The steering file ended when more input was required.

Es010: unexpected end of line

A line in the steering file ended before the directive was complete.

Es011: unexpected text after end of directive

There is more text on the same line after the end of a steering file directive.

Es012: expected text

The specified text was not present in the steering file, but must be present for the directive to be correct.

Es013: pattern can contain at most one * or ?

Each pattern in the current directive can contain at most one * or one ? wildcard character.

Es014: rename patterns have different wildcards

Both patterns in the current directive must contain exactly the same kind of wildcard. That is, both must either contain:

- No wildcards
- Exactly one *
- Exactly one ?

This error occurs if the patterns are not the same in this regard.

Es014: ambiguous pattern match: *symbol* matches more than one rename pattern

A symbol in the input file matches more than one `rename` pattern.

Descriptions of options

This section gives detailed reference information about each command line option available for the different utilities.

--all

Syntax	--all
Tool	ielfdumparm
Description	Use this option to include the contents of all ELF sections in the output, in addition to the general properties of the input file. Sections are output in index order, except that each relocation section is output immediately after the section it holds relocations for. By default, no section contents are included in the output.



This option is not available in the IDE.

--bin

Syntax	--bin
Tool	ielftool
Description	Sets the format of the output file to binary.



To set related options, choose:

Project>Options>Output converter

--checksum

Syntax	<code>--checksum {symbol[+offset] address}:size,algorithm[:[1 2][m][r][i p]][,start];range[:range...]</code>	
Parameters	<i>symbol</i>	The name of the symbol where the checksum value should be stored. Note that it must exist in the symbol table in the input ELF file.
	<i>offset</i>	An offset to the symbol.

	<i>address</i>	The absolute address where the checksum value should be stored.
	<i>size</i>	The number of bytes in the checksum: 1, 2, or 4; must not be larger than the size of the checksum symbol.
	<i>algorithm</i>	<p>The checksum algorithm used, one of:</p> <ul style="list-style-type: none">• <i>sum</i>, a byte-wise calculated arithmetic sum. The result is truncated to 8 bits.• <i>sum8wide</i>, a byte-wise calculated arithmetic sum. The result is truncated to the size of the symbol.• <i>sum32</i>, a word-wise (32 bits) calculated arithmetic sum• <i>crc16</i>, CRC16 (generating polynomial 0x11021); used by default• <i>crc32</i>, CRC32 (generating polynomial 0x104C11DB7)• <i>crc=n</i>, CRC with a generating polynomial of <i>n</i>.
	<i>1 2</i>	<p>If specified, can be one of:</p> <ul style="list-style-type: none">• 1 – Specifies one's complement.• 2 – Specifies two's complement.
	<i>m</i>	Reverses the order of the bits within each byte when calculating the checksum.
	<i>r</i>	Reverses the byte order of the input data within each word of size <i>size</i> .
	<i>i p</i>	<p>Use either <i>i</i> or <i>p</i>, if the <i>start</i> value is bigger than 0. If specified, can be one of:</p> <ul style="list-style-type: none">• <i>i</i> – Initializes the checksum value with the start value.• <i>p</i> – Prefixes the input data with a word of size <i>size</i> that contains the <i>start</i> value.
	<i>start</i>	By default, the initial value of the checksum is 0. If necessary, use <i>start</i> to supply a different initial value. If not 0, then either <i>i</i> or <i>p</i> must be specified.
	<i>range</i>	<p>The address range on which the checksum should be calculated.</p> <p>Hexadecimal and decimal notation is allowed (for example, 0x8002–0x8FFF).</p>
Tool	<i>ielftool</i>	
Description		Use this option to calculate a checksum with the specified algorithm for the specified ranges. The checksum will then replace the original value in <i>symbol</i> . A new absolute symbol will be generated; with the <i>symbol</i> name suffixed with <i>_value</i> containing the calculated checksum. This symbol can be used for accessing the checksum value later when needed, for example during debugging.

If the `--checksum` option is used more than once on the command line, the options are evaluated from left to right. If a checksum is calculated for a *symbol* that is specified in a later evaluated `--checksum` option, an error is issued.

Example

This example shows how to use the `crc16` algorithm with the start value 0 over the address range 0x8000–0x8FFF:

```
ielftool --checksum=__checksum:2,crc16;0x8000-0x8FFF
sourceFile.out destinationFile.out
```

The input data is read from `sourceFile.out`, and the resulting checksum value of size 2 bytes will be stored at the symbol `__checksum`. The modified ELF file is saved as `destinationFile.out` leaving `sourceFile.out` untouched.

See also

Checksum calculation, page 177



To set related options, choose:

Project>Options>Linker>Checksum

--code

Syntax

`--code`

Tool

`ielfdump`

Description

Use this option to dump all sections that contain executable code (sections with the ELF section attribute `SHF_EXECINSTR`).



This option is not available in the IDE.

--create

Syntax

`--create libraryfile objectfile1 ... objectfileN`


Parameters

libraryfile The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters, page 234*.


objectfile1 ... objectfileN The object file(s) to build the library from.

Tool


`iarchive`

Description	Use this command to build a new library from a set of object files (modules). The object files are added to the library in the exact order that they are specified on the command line. If no command is specified on the command line, <code>--create</code> is used by default.
	 This option is not available in the IDE.

--delete, -d

Syntax	<code>--delete libraryfile objectfile1 ... objectfileN</code> <code>-d libraryfile objectfile1 ... objectfileN</code>
Parameters	<p><i>libraryfile</i> The library file that the command operates on. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i>, page 234.</p> <p><i>objectfile1 ... objectfileN</i> The object file(s) that the command operates on.</p>
Tool	iarchive
Description	Use this command to remove object files (modules) from an existing library. All object files that are specified on the command line will be removed from the library.
	 This option is not available in the IDE.

--edit

Syntax	<code>--edit steering_file</code>
Tool	isymexport
Description	Use this option to specify a steering file to control which symbols that are included in the <code>isymexport</code> output file, and also to rename some of the symbols if that is desired.
See also	<i>Steering files</i> , page 427.
	 This option is not available in the IDE.

--extract, -x

Syntax	<pre>--extract libraryfile [objectfile1 ... objectfileN] -x libraryfile [objectfile1 ... objectfileN]</pre>
Parameters	<p><i>libraryfile</i> The library file that the command operates on. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i>, page 234.</p> <p><i>objectfile1 ... objectfileN</i> The object file(s) that the command operates on.</p>
Tool	iarchive
Description	Use this command to extract object files (modules) from an existing library. If a list of object files is specified, only these files are extracted. If a list of object files is not specified, all object files in the library are extracted.



This option is not available in the IDE.

-f

Syntax	<pre>-f filename</pre>
Parameters	For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 234.
Tool	iarchive, ielfdumparm, iobjmanip, and isymexport.
Description	<p>Use this option to make the tool read command line options from the named file, with the default filename extension <code>.xcl</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you can use multiple lines, because the newline character acts just as a space or tab character.</p> <p>Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.</p>



This option is not available in the IDE.

--fill

Syntax	<code>--fill <i>pattern</i>;<i>range</i>[;<i>range</i>...]</code>	
Parameters	<i>range</i>	Specifies the address range for the fill. Hexadecimal and decimal notation is allowed (for example, 0x8002–0x8FFF). Note that each address must be 4-byte aligned.
	<i>pattern</i>	A hexadecimal string with the 0x prefix (for example, 0xEF) interpreted as a sequence of bytes, where each pair of digits corresponds to one byte (for example 0x123456, for the sequence of bytes 0x12, 0x34, and 0x56). This sequence is repeated over the fill area. If the length of the fill pattern is greater than 1 byte, it is repeated as if it started at address 0.
Applicability	ielftool	
Description	Use this option to fill all gaps in one or more ranges with a pattern, which can be either an expression or a hexadecimal string. The contents will be calculated as if the fill pattern was repeatedly filled from the start address until the end address is passed, and then the real contents will overwrite that pattern.	
	If the <code>--fill</code> option is used more than once on the command line, the fill ranges cannot overlap each other.	



To set related options, choose:
Project>Options>Linker>Checksum

--ihex

Syntax	<code>--ihex</code>
Tool	ielftool
Description	Sets the format of the output file to linear Intel hex.



To set related options, choose:
Project>Options>Linker>Output converter

--no_strtab

Syntax	<code>--no_strtab</code>
Tool	<code>ielfdumparm</code>
Description	Use this option to suppress dumping of string table sections (sections of type <code>SHT_STRTAB</code>).



This option is not available in the IDE.

--output, -o

Syntax	<code>-o {filename directory}</code> <code>--output {filename directory}</code>
Parameters	For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 234.
Tool	<code>iarchive</code> and <code>ielfdumparm</code> .
Description	<p><code>iarchive</code></p> <p>By default, <code>iarchive</code> assumes that the first argument after the <code>iarchive</code> command is the name of the destination library. Use this option to explicitly specify a different filename for the library.</p> <p><code>ielfdumparm</code></p> <p>By default, output from the dumper is directed to the console. Use this option to direct the output to a file instead. The default name of the output file is the name of the input file with an added <code>id</code> filename extension</p> <p>You can also specify the output file by specifying a file or directory following the name of the input file.</p>



This option is not available in the IDE.

--ram_reserve_ranges

Syntax	<code>--ram_reserve_ranges [=symbol_prefix]</code>
Parameters	<p><code>symbol_prefix</code> The prefix of symbols created by this option.</p>

Tool	<code>isymexport</code>
Description	<p>Use this option to generate symbols for the areas in RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter <i>symbol_prefix</i>.</p> <p>Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.</p> <p>If <code>--ram_reserve_ranges</code> is used together with <code>--reserve_ranges</code>, the RAM areas will get their prefix from the <code>--ram_reserve_ranges</code> option and the non-RAM areas will get their prefix from the <code>--reserve_ranges</code> option.</p>
See also	<code>--reserve_ranges</code> , page 440.



This option is not available in the IDE.

--raw

Syntax	<code>--raw</code>
Tool	<code>ielfdumparm</code>
Description	<p>By default, many ELF sections will be dumped using a text format specific to a particular kind of section. Use this option to dump each selected ELF section using the generic text format.</p> <p>The generic text format dumps each byte in the section in hexadecimal format, and where appropriate, as ASCII text.</p>



This option is not available in the IDE.

--remove_section

Syntax	<code>--remove_section {<i>section</i> <i>number</i>}</code>				
Parameters	<table><tr><td><i>section</i></td><td>The section—or sections, if there are more than one section with the same name—to be removed.</td></tr><tr><td><i>number</i></td><td>The number of the section to be removed. Section numbers can be obtained from an object dump created using <code>ielfdumparm</code>.</td></tr></table>	<i>section</i>	The section—or sections, if there are more than one section with the same name—to be removed.	<i>number</i>	The number of the section to be removed. Section numbers can be obtained from an object dump created using <code>ielfdumparm</code> .
<i>section</i>	The section—or sections, if there are more than one section with the same name—to be removed.				
<i>number</i>	The number of the section to be removed. Section numbers can be obtained from an object dump created using <code>ielfdumparm</code> .				

Tool	iobjmanip
Description	Use this option to make <code>iobjmanip</code> omit the specified section when generating the output file.



This option is not available in the IDE.

--rename_section

Syntax	<code>--rename_section {<i>oldname</i> <i>oldnumber</i>}=<i>newname</i></code>	
Parameters	<i>oldname</i>	The section—or sections, if there are more than one section with the same name—to be renamed.
	<i>oldnumber</i>	The number of the section to be renamed. Section numbers can be obtained from an object dump created using <code>ielfdumparm</code> .
	<i>newname</i>	The new name of the section.
Tool	iobjmanip	
Description	Use this option to make <code>iobjmanip</code> rename the specified section when generating the output file.	



This option is not available in the IDE.

--rename_symbol

Syntax	<code>--rename_symbol <i>oldname</i> =<i>newname</i></code>	
Parameters	<i>oldname</i>	The symbol to be renamed.
	<i>newname</i>	The new name of the symbol.
Tool	iobjmanip	
Description	Use this option to make <code>iobjmanip</code> rename the specified symbol when generating the output file.	



This option is not available in the IDE.

--replace, -r

Syntax	<pre>--replace libraryfile objectfile1 ... objectfileN -r libraryfile objectfile1 ... objectfileN</pre>	
Parameters	<i>libraryfile</i>	The library file that the command operates on. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 234.
	<i>objectfile1</i> ... <i>objectfileN</i>	The object file(s) that the command operates on.
Tool	iarchive	
Description	Use this command to replace or add object files (modules) to an existing library. The object files specified on the command line either replace existing object files in the library (if they have the same name) or are appended to the library.	



This option is not available in the IDE.

--reserve_ranges

Syntax	<pre>--reserve_ranges[=<i>symbol_prefix</i>]</pre>	
Parameters	<i>symbol_prefix</i>	The prefix of symbols created by this option.
Tool	isymexport	
Description	<p>Use this option to generate symbols for the areas in ROM and RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter <i>symbol_prefix</i>.</p> <p>Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.</p> <p>If <code>--reserve_ranges</code> is used together with <code>--ram_reserve_ranges</code>, the RAM areas will get their prefix from the <code>--ram_reserve_ranges</code> option and the non-RAM areas will get their prefix from the <code>--reserve_ranges</code> option.</p>	

See also

--ram_reserve_ranges, page 437.



This option is not available in the IDE.

--section, -s

Syntax

```
--section section_number|section_name[,...]
--s section_number|section_name[,...]
```

Parameters

section_number The number of the section to be dumped.
section_name The name of the section to be dumped.

Tool

ielfdumparm

Description

Use this option to dump the contents of a section with the specified number, or any section with the specified name. If a relocation section is associated with a selected section, its contents are output as well.

If you use this option, the general properties of the input file will not be included in the output.

You can specify multiple section numbers or names by separating them with commas, or by using this option more than once.

By default, no section contents are included in the output.

Example

```
-s 3,17                    /* Sections #3 and #17
-s .debug_frame,42        /* Any sections named .debug_frame and
                            also section #42 */
```



This option is not available in the IDE.

--self_reloc

Syntax

```
--self_reloc
```

Tool

ielftool


Description

This option is intentionally not documented as it is not intended for general use.




This option is not available in the IDE.


--silent

Syntax	--silent -S (iarchive only)
Tool	iarchive and ielftool.
Description	<p>Causes the tool to operate without sending any messages to the standard output stream.</p> <p>By default, ielftool sends various messages via the standard output stream. You can use this option to prevent this. ielftool sends error and warning messages to the error output stream, so they are displayed regardless of this setting.</p> <p> This option is not available in the IDE.</p>


--simple

Syntax	--simple
Tool	ielftool
Description	<p>Sets the format of the output file to Simple code.</p> <p> To set related options, choose: Project>Options>Output converter</p>


--srec

Syntax	--srec
Tool	ielftool
Description	<p>Sets the format of the output file to Motorola S-records.</p> <p> To set related options, choose: Project>Options>Output converter</p>


--srec-len

Syntax	<code>--srec-len=length</code>	
Parameters	<i>length</i>	The number of data bytes in each S-record.
Tool	<code>ielftool</code>	
Description	Restricts the number of data bytes in each S-record. This option can be used in combination with the <code>--srec</code> option.	
		This option is not available in the IDE.


--srec-s3only

Syntax	<code>--srec-s3only</code>	
Tool	<code>ielftool</code>	
Description	Restricts the S-record output to contain only a subset of records, that is S0, S3 and S7 records. This option can be used in combination with the <code>--srec</code> option.	
		This option is not available in the IDE.


--strip

Syntax	<code>--strip</code>	
Tool	<code>iobjmanip</code> and <code>ielftool</code> .	
Description	Use this option to remove all sections containing debug information before the output file is written.	
	Note that <code>ielftool</code> needs an unstripped input ELF image. If you use the <code>--strip</code> option in the linker, remove it and use the <code>--strip</code> option in <code>ielftool</code> instead.	
		To set related options, choose: Project>Options>Linker>Output>Include debug information in output

--symbols

Syntax	<code>--symbols libraryfile</code>	
Parameters	<i>libraryfile</i>	The library file that the command operates on. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters, page 234</i> .
Tool	iarchive	
Description	<p>Use this command to list all external symbols that are defined by any object file (module) in the specified library, together with the name of the object file (module) that defines it.</p> <p>In silent mode (<code>--silent</code>), this command performs symbol table-related syntax checks on the library file and displays only errors and warnings.</p>	
		This option is not available in the IDE.

--toc, -t

Syntax	<code>--toc libraryfile</code> <code>-t libraryfile</code>	
Parameters	<i>libraryfile</i>	The library file that the command operates on. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters, page 234</i> .
Tool	iarchive	
Description	<p>Use this command to list the names of all object files (modules) in a specified library.</p> <p>In silent mode (<code>--silent</code>), this command performs basic syntax checks on the library file, and displays only errors and warnings.</p>	
		This option is not available in the IDE.

--verbose, -V

Syntax	<code>--verbose</code> <code>-V (iarchive only)</code>
--------	---

Tool	<code>iarchive</code> and <code>ielftool</code> .
Description	Use this option to make the tool report which operations it performs, in addition to giving diagnostic messages.
	 This option is not available in the IDE because this setting is always enabled.

Implementation-defined behavior for Standard C

This chapter describes how the compiler handles the implementation-defined areas of the C language based on Standard C.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

If you are using C89 instead of Standard C, see *Implementation-defined behavior for C89, page 461*. For a short overview of the differences between Standard C and C89, see *C language overview, page 151*.

Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

J.3.1 Translation

Diagnostics (3.10, 5.1.1.3)

Diagnostics are produced in the form:

filename, *linenumber* *level*[*tag*]: *message*

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.

J.3.2 Environment

The character set (5.1.1.2)

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, if you use the `--enable_multibytes` compiler option, the host character set is used instead.

Main (5.1.2.1)

The function called at program startup is called `main`. No prototype is declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior, see *Customizing system initialization, page 111*.

The effect of program termination (5.1.2.1)

Terminating the application returns the execution to the startup code (just after the call to `main`).

Alternative ways to define main (5.1.2.2.1)

There is no alternative ways to define the `main` function.

The argv argument to main (5.1.2.2.1)

The `argv` argument is not supported.

Streams as interactive devices (5.1.2.3)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

Signals, their semantics, and the default handling (7.14)

In the DLIB library, the set of supported signals is the same as in Standard C. A raised signal will do nothing, unless the `signal` function is customized to fit the application.

Signal values for computational exceptions (7.14.1.1)

In the DLIB library, there are no implementation-defined values that correspond to a computational exception.

Signals at system startup (7.14.1.1)

In the DLIB library, there are no implementation-defined signals that are executed at system startup.

Environment names (7.20.4.5)

In the DLIB library, there are no implementation-defined environment names that are used by the `getenv` function.

The system function (7.20.4.6)

The `system` function is not supported.

J.3.3 Identifiers**Multibyte characters in identifiers (6.4.2)**

Additional multibyte characters may not appear in identifiers.

Significant characters in identifiers (5.2.4.1, 6.1.2)

The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

J.3.4 Characters**Number of bits in a byte (3.6)**

A byte contains 8 bits.

Execution character set member values (5.2.1)

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the host character set.

Alphabetic escape sequences (5.2.2)

The standard alphabetic escape sequences have the values `\a-7`, `\b-8`, `\f-12`, `\n-10`, `\r-13`, `\t-9`, and `\v-11`.

Characters outside of the basic executive character set (6.2.5)

A character outside of the basic executive character set that is stored in a `char` is not transformed.

Plain char (6.2.5, 6.3.1.1)

A plain `char` is treated as an `unsigned char`.

Source and execution character sets (6.4.4.4, 5.1.1.2)

The source character set is the set of legal characters that can appear in source files. By default, the source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. By default, the execution character set is the standard ASCII character set.

However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 117.

Integer character constants with more than one character (6.4.4.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

Wide character constants with more than one character (6.4.4.4)

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Locale used for wide character constants (6.4.4.4)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Locale used for wide string literals (6.4.5)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Source characters as executive characters (6.4.5)

All source characters can be represented as executive characters.

J.3.5 Integers

Extended integer types (6.2.5)

There are no extended integer types.

Range of integer values (6.2.6.2)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

For information about the ranges for the different integer types, see *Basic data types*, page 288.

The rank of extended integer types (6.3.1.1)

There are no extended integer types.

Signals when converting to a signed integer type (6.3.1.3)

No signal is raised when an integer is converted to a signed integer type.

Signed bitwise operations (6.5)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

J.3.6 Floating point

Accuracy of floating-point operations (5.2.4.2.2)

The accuracy of floating-point operations is unknown.

Rounding behaviors (5.2.4.2.2)

There are no non-standard values of `FLT_ROUNDS`.

Evaluation methods (5.2.4.2.2)

There are no non-standard values of `FLT_EVAL_METHOD`.

Converting integer values to floating-point values (6.3.1.4)

When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Converting floating-point values to floating-point values (6.3.1.5)

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Denoting the value of floating-point constants (6.4.4.2)

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Contraction of floating-point values (6.5)

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

Default state of `FENV_ACCESS` (7.6.1)

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

Additional floating-point mechanisms (7.6, 7.12)

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

Default state of `FP_CONTRACT` (7.12.2)

The default state of the pragma directive `FP_CONTRACT` is `OFF`.

J.3.7 Arrays and pointers

Conversion from/to pointers (6.3.2.3)

For information about casting of data pointers and function pointers, see *Casting*, page 296.

`ptrdiff_t` (6.5.6)

For information about `ptrdiff_t`, see *ptrdiff_t*, page 297.

J.3.8 Hints

Honoring the register keyword (6.7.1)

User requests for register variables are not honored.

Inlining functions (6.7.4)

User requests for inlining functions increases the chance, but does not make it certain, that the function will actually be inlined into another function. See the pragma directive *inline*, page 324.

J.3.9 Structures, unions, enumerations, and bitfields**Sign of 'plain' bitfields (6.7.2, 6.7.2.1)**

For information about how a 'plain' `int` bitfield is treated, see *Bitfields*, page 290.

Possible types for bitfields (6.7.2.1)

All integer types can be used as bitfields in the compiler's extended mode, see *-e*, page 235.

Bitfields straddling a storage-unit boundary (6.7.2.1)

A bitfield is always placed in one—and one only—storage unit, which means that the bitfield cannot straddle a storage-unit boundary.

Allocation order of bitfields within a unit (6.7.2.1)

For information about how bitfields are allocated within a storage unit, see *Bitfields*, page 290.

Alignment of non-bitfield structure members (6.7.2.1)

The alignment of non-bitfield members of structures is the same as for the member types, see *Alignment*, page 287.

Integer type used for representing enumeration types (6.7.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

J.3.10 Qualifiers**Access to volatile objects (6.7.3)**

Any reference to an object with `volatile` qualified type is an access, see *Declaring objects volatile*, page 299.

J.3.11 Preprocessing directives

Mapping of header names (6.4.7)

Sequences in header names are mapped to source file names verbatim. A backslash '\' is not treated as an escape sequence. See *Overview of the preprocessor*, page 367.

Character constants in constant expressions (6.10.1)

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

The value of a single-character constant (6.10.1)

A single-character constant may only have a negative value if a plain character (`char`) is treated as a signed character, see *--char_is_signed*, page 227.

Including bracketed filenames (6.10.2)

For information about the search algorithm used for file specifications in angle brackets `<>`, see *Include file search procedure*, page 211.

Including quoted filenames (6.10.2)

For information about the search algorithm used for file specifications enclosed in quotes, see *Include file search procedure*, page 211.

Preprocessing tokens in `#include` directives (6.10.2)

Preprocessing tokens in an `#include` directive are combined in the same way as outside an `#include` directive.

Nesting limits for `#include` directives (6.10.2)

There is no explicit nesting limit for `#include` processing.

Universal character names (6.10.3.2)

Universal character names (UCN) are not supported.

Recognized pragma directives (6.10.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter overrides the information here.

`alignment`

baseaddr
basic_template_matching
building_runtime
can_instantiate
codeseg
constseg
cspy_support
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_requirement_override
library_provides
library_requirement_override
memory
module_name
no_pch
once
public_equ
system_include
vector
warnings

Default `__DATE__` and `__TIME__` (6.10.8)

The definitions for `__TIME__` and `__DATE__` are always available.

J.3.12 Library functions

Additional library facilities (5.1.2.1)

Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 93.

Diagnostic printed by the assert function (7.2.1.1)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

Representation of the floating-point status flags (7.6.2.2)

For information about the floating-point status flags, see *fenv.h*, page 379.

Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 294.

Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 117.

Types defined for float_t and double_t (7.12)

The `FLT_EVAL_METHOD` macro can only have the value 0.

Domain errors (7.12.1)

No function generates other domain errors than what the standard requires.

Return values on domain errors (7.12.1)

Mathematic functions return a floating-point NaN (not a number) for domain errors.

Underflow errors (7.12.1)

Mathematic functions set `errno` to the macro `ERANGE` (a macro in `errno.h`) and return zero for underflow errors.

fmod return value (7.12.10.1)

The `fmod` function returns a floating-point NaN when the second argument is zero.

The magnitude of remquo (7.12.10.3)

The magnitude is congruent modulo `INT_MAX`.

signal() (7.14.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 121.

NULL macro (7.17)

The `NULL` macro is defined to 0.

Terminating newline character (7.19.2)

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

Space characters before a newline character (7.19.2)

Space characters written to a stream immediately before a newline character are preserved.

Null characters appended to data written to binary streams (7.19.2)

No null characters are appended to data written to binary streams.

File position in append mode (7.19.3)

The file position is initially placed at the beginning of the file when it is opened in append-mode.

Truncation of files (7.19.3)

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 117.

File buffering (7.19.3)

An open file can be either block-buffered, line-buffered, or unbuffered.

A zero-length file (7.19.3)

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

Legal file names (7.19.3)

The legality of a filename depends on the application-specific implementation of the low-level file routines.

Number of times a file can be opened (7.19.3)

Whether a file can be opened more than once depends on the application-specific implementation of the low-level file routines.

Multibyte characters in a file (7.19.3)

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

remove() (7.19.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 117.

rename() (7.19.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 117.

Removal of open temporary files (7.19.4.3)

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

Mode changing (7.19.5.4)

`freopen` closes the named stream, then reopens it in the new mode. The streams `stdin`, `stdout`, and `stderr` can be reopened in any new mode.

Style for printing infinity or NaN (7.19.6.1, 7.24.2.1)

The style used for printing infinity or NaN for a floating-point constant is `inf` and `nan` (`INF` and `NAN` for the `F` conversion specifier), respectively. The `n-char-sequence` is not used for `nan`.

%p in printf() (7.19.6.1, 7.24.2.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

Reading ranges in scanf (7.19.6.2, 7.24.2.1)

A `-` (dash) character is always treated as a range symbol.

%p in scanf (7.19.6.2, 7.24.2.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

File position errors (7.19.9.1, 7.19.9.3, 7.19.9.4)

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)

An `n`-char-sequence after a NaN is read and ignored.

errno value at underflow (7.20.1.3, 7.24.4.1.1)

`errno` is set to `ERANGE` if an underflow is encountered.

Zero-sized heap objects (7.20.3)

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

Behavior of abort and exit (7.20.4.1, 7.20.4.4)

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)

The termination status will be propagated to `__exit()` as a parameter. `exit()` and `_Exit()` use the input parameter, whereas `abort` uses `EXIT_FAILURE`.

The system function return value (7.20.4.6)

The `system` function is not supported.

The time zone (7.23.1)

The local time zone and daylight savings time must be defined by the application. For more information, see *Time*, page 121.

Range and precision of time (7.23)

The implementation uses `signed long` for representing `clock_t` and `time_t`, based at the start of the year 1970. This gives a range of approximately plus or minus 69 years in seconds. However, the application must supply the actual implementation for the functions `time` and `clock`. See *Time*, page 121.

clock() (7.23.2.1)

The application must supply an implementation of the `clock` function. See *Time*, page 121.

%Z replacement string (7.23.3.5, 7.24.5.1)

By default, ":" is used as a replacement for %Z. Your application should implement the time zone handling. See *Time*, page 121.

Math functions rounding mode (F.9)

The functions in `math.h` honor the rounding direction mode in `FLT-ROUNDS`.

J.3.13 Architecture

Values and expressions assigned to some macros (5.2.4.2, 7.18.2, 7.18.3)

There are always 8 bits in a byte.

`MB_LEN_MAX` is at the most 6 bytes depending on the library configuration that is used.

For information about sizes, ranges, etc for all basic types, see *Data representation*, page 287.

The limit macros for the exact-width, minimum-width, and fastest minimum-width integer types defined in `stdint.h` have the same ranges as `char`, `short`, `int`, `long`, and `long long`.

The floating-point constant `FLT_ROUNDS` has the value 1 (to nearest) and the floating-point constant `FLT_EVAL_METHOD` has the value 0 (treat as is).

The number, order, and encoding of bytes (6.2.6.1)

See *Data representation*, page 287.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 287.

J.4 Locale**Members of the source and execution character set (5.2.1)**

By default, the compiler accepts all one-byte characters in the host's default character set. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

The meaning of the additional character set (5.2.1.2)

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

Shift states for encoding multibyte characters (5.2.1.2)

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

Direction of successive printing characters (5.2.2)

The application defines the characteristics of a display device.

The decimal point character (7.1.1)

The default decimal-point character is a `'.'`. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

Printing characters (7.4, 7.25.2)

The set of printing characters is determined by the chosen locale.

Control characters (7.4, 7.25.2)

The set of control characters is determined by the chosen locale.

Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

The sets of characters tested are determined by the chosen locale.

The native environment (7.1.1.1)

The native environment is the same as the "C" locale.

Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

The collation of the execution character set (7.21.4.3, 7.24.4.4.2)

The collation of the execution character set is determined by the chosen locale.

Message returned by strerror (7.21.6.2)

The messages returned by the `strerror` function depending on the argument is:

Argument	Message
EZERO	no error
EDOM	domain error
ERANGE	range error
EFPOS	file positioning error
EILSEQ	multi-byte encoding error
<0 >99	unknown error
all others	error <i>nnn</i>

Table 54: Message returned by `strerror()`—IAR DLIB library

Implementation-defined behavior for C89

This chapter describes how the compiler handles the implementation-defined areas of the C language based on the C89 standard.

If you are using Standard C instead of C89, see *Implementation-defined behavior for Standard C*, page 445. For a short overview of the differences between Standard C and C89, see *C language overview*, page 151.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

Translation

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

filename,linenumber level[tag]: message

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

Environment

Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```


To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 111.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

Identifiers

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

Characters

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 117.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 117.

Range of 'plain' char (6.2.1.1)

A ‘plain’ char has the same range as an unsigned char.

Integers**Range of integer values (6.1.2.5)**

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types*, page 288, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

Floating point

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (*s*), a biased exponent (*e*), and a mantissa (*m*).

See *Floating-point types*, page 294, for information about the ranges and sizes for the different floating-point types: `float` and `double`.

Converting integer values to floating-point values (6.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

Demoting floating-point values (6.2.1.4)

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

Arrays and pointers

size_t (6.3.3.4, 7.1.1)

See *size_t*, page 296, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 296, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 297, for information about the `ptrdiff_t`.

Registers

Honoring the register keyword (6.5.1)

User requests for register variables are not honored.

Structures, unions, enumerations, and bitfields

Improper access to a union (6.3.2.3)

If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

Padding and alignment of structure members (6.5.2.1)

See the section *Basic data types*, page 288, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a `unsigned int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

Qualifiers

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

Declarators

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

Statements

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

Preprocessing directives

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a signed character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file is not found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter overrides the information here.

```
alignment
baseaddr
basic_template_matching
building_runtime
can_instantiate
codeseg
constseg
cspy_support
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
```

```

keep_definition
library_requirement_override
library_provides
library_requirement_override
memory
module_name
no_pch
once
public_equ
system_include
vector
warnings

```

Default `__DATE__` and `__TIME__` (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB Library functions

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to 0.

Diagnostic printed by the `assert` function (7.2)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

Domain errors (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns NaN; `errno` is set to EDOM.

signal() (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 121.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

No null characters are appended to data written to binary streams.

Files (7.9.3)

Whether the file position indicator of an append-mode stream is initially positioned at the beginning or the end of the file, depends on the application-specific implementation of the low-level file routines.

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 117.

The characteristics of the file buffering is that the implementation supports files that are unbuffered, line buffered, or fully buffered.

Whether a zero-length file actually exists depends on the application-specific implementation of the low-level file routines.

Rules for composing valid file names depends on the application-specific implementation of the low-level file routines.

Whether the same file can be simultaneously open multiple times depends on the application-specific implementation of the low-level file routines.

remove() (7.9.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 117.

rename() (7.9.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 117.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

Message generated by perror() (7.9.10.4)

The generated message is:

usersuppliedprefix:errormessage

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of `exit()` (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 120.

`system()` (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 120.

Message returned by `strerror()` (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

Argument	Message
EZERO	no error
EDOM	domain error
ERANGE	range error
EFPOS	file positioning error
EILSEQ	multi-byte encoding error
<0 >99	unknown error
all others	error <i>nnn</i>

Table 55: Message returned by `strerror()`—IAR DLIB library

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 121.

`clock()` (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 121.

A

__AAPCS__ (predefined symbol) 368
 --aapcs (compiler option) 226
 __AAPCS_VFP__ (predefined symbol) 368
 ABI, AEABI and IA64 182
 abort
 implementation-defined behavior. 457
 implementation-defined behavior in C89 (DLIB) . . . 470
 system termination (DLIB) 110
 Abort_Handler (exception function) 60
 __absolute (extended keyword). 307
 absolute location
 data, placing at (@) 192
 language support for 154
 placing data in registers (@) 194
 #pragma location 326
 --aeabi (compiler option) 226
 __AEABI_PORTABILITY_LEVEL (preprocessor
 symbol). 184
 __AEABI_PORTABLE (preprocessor symbol) 184
 algorithm (STL header file) 377
 alignment 287
 forcing stricter (#pragma data_alignment) 321
 in structures (#pragma pack) 329
 in structures, causing problems 189
 of an object (__ALIGNOF__) 154
 of data types. 287
 restrictions for inline assembler 131
 alignment (pragma directive) 452, 467
 __ALIGNOF__ (operator) 154
 --align_sp_on_irq (compiler option). 226
 --all (ielfdump option) 431
 anonymous structures 189
 anonymous symbols, creating 151
 ANSI C. *See* C89
 application
 building, overview of 47
 execution, overview of 42

 startup and termination (DLIB) 107
 architecture
 more information about 21
 of ARM 53
 argv (argument), implementation-defined behavior 446
 ARM
 and Thumb code, overview 57
 CPU mode 48
 memory layout. 53
 __arm (extended keyword) 307
 --arm (compiler option). 227
 __ARMVFP__ (predefined symbol). 368
 __ARMVFPV2__ (predefined symbol) 368
 __ARMVFPV3__ (predefined symbol) 368
 __ARMVFPV4__ (predefined symbol) 368
 __ARMVFP_D6__ (predefined symbol) 368
 __ARMVFP_FP16__ (predefined symbol). 368
 __ARMVFP_SP__ (predefined symbol). 368
 __ARM_ADVANCED_SIMD__ (predefined symbol) . . . 368
 __ARM_MEDIA__ (predefined symbol) 368
 __ARM_PROFILE_M__ (predefined symbol). 368
 __ARM4M__ (predefined symbol). 369
 __ARM4TM__ (predefined symbol) 369
 __ARM5__ (predefined symbol) 369
 __ARM5E__ (predefined symbol) 369
 __ARM6__ (predefined symbol) 369
 __ARM6M__ (predefined symbol). 369
 __ARM6SM__ (predefined symbol). 369
 __ARM7A__ (predefined symbol) 369
 __ARM7EM__ (predefined symbol) 369
 __ARM7M__ (predefined symbol). 369
 __ARM7R__ (predefined symbol) 369
 arrays
 designated initializers in 151
 implementation-defined behavior. 450
 implementation-defined behavior in C89 465
 incomplete at end of structs 151
 non-lvalue 157
 of incomplete types 156

single-value initialization	158
asm, __asm (language extension)	132
assembler code	
calling from C	136
calling from C++	138
inserting inline	131
assembler directives	
for call frame information	146
using in inline assembler code	131
assembler instructions	
for software interrupts	62
inserting inline	131
assembler labels, making public (--public_equ)	253
assembler language interface	129
calling convention. <i>See</i> assembler code	
assembler list file, generating	240
assembler output file	138
asserts	122
implementation-defined behavior of	454
implementation-defined behavior of in C89, (DLIB)	468
including in application	371
assert.h (DLIB header file)	375
__assignment_by_bitwise_copy_allowed, symbol used	
in library	381
@ (operator)	
placing at absolute address	192
placing in sections	193
atexit	122
reserving space for calls	86
atexit limit, setting up	86
attributes	
object	305
type	303
auto variables	54
at function entrance	141
programming hints for efficient code	200
using in inline assembler statements	132
auto, packing algorithm for initializers	392

B

backtrace information <i>See</i> call frame information	
Barr, Michael	25
baseaddr (pragma directive)	453, 467
__BASE_FILE__ (predefined symbol)	369
basic type names, using in preprocessor expressions	
(--migration_preprocessor_extensions)	242
--basic_heap (linker option)	264
basic_template_matching (pragma directive)	453, 467
batch files	
error return codes	213
none for building library from command line	106
--BE8 (linker option)	264
--BE32 (linker option)	264
__big_endian (extended keyword)	307
big-endian (byte order)	49
--bin (ielftool option)	431
binary streams	455
binary streams in C89 (DLIB)	469
bit negation	202
bitfields	
data representation of	290
hints	188
implementation-defined behavior	451
implementation-defined behavior in C89	465
non-standard types in	154
bitfields (pragma directive)	319
bits in a byte, implementation-defined behavior	447
bitset (library header file)	377
bold style, in this guide	26
bool (data type)	288
adding support for in DLIB	376, 379
building_runtime (pragma directive)	453, 467
__BUILD_NUMBER__ (predefined symbol)	369
Burrows-Wheeler algorithm, for packing initializers	392
BusFault_Handler (exception function)	64
bwt, packing algorithm for initializers	392

byte order	49
identifying	370
specifying (<code>--endian</code>)	236

C

C and C++ linkage	140
C/C++ calling convention. <i>See</i> calling convention	
C header files	375
C language, overview	151
call frame information	146
in assembler list file	138
in assembler list file (<code>-IA</code>)	240
call graph root (stack usage control directive)	413
call stack	146
callee-save registers, stored on stack	54
calling convention	
C++, requiring C linkage	138
in compiler	139
calloc (library function)	55
<i>See also</i> heap	
implementation-defined behavior in C89 (DLIB)	470
calls (pragma directive)	320
<code>--call_graph</code> (compiler option)	265
<code>call_graph_root</code> (pragma directive)	320
<code>call-info</code> (in stack usage control file)	414
<code>can_instantiate</code> (pragma directive)	453, 467
cassert (library header file)	378
cast operators	
in Extended EC++	162, 169
missing from Embedded C++	162
casting	
of pointers and integers	296
pointers to integers, language extension	157
category (in stack usage control file)	413
ccomplex (library header file)	378
cctype (DLIB header file)	378
cerrno (DLIB header file)	378
cexit (system termination code)	107
cfenv (library header file)	378
CFI (assembler directive)	146
CFI_COMMON_ARM (CFI macro)	148
CFI_COMMON_Thumb (CFI macro)	148
CFI_NAME_BLOCK (CFI macro)	148
cfloat (DLIB header file)	378
char (data type)	288
changing default representation (<code>--char_is_signed</code>)	227
changing representation (<code>--char_is_unsigned</code>)	228
implementation-defined behavior	447
signed and unsigned	289
character set, implementation-defined behavior	446
characters, implementation-defined behavior	447
characters, implementation-defined behavior in C89	462
character-based I/O	113
<code>--char_is_signed</code> (compiler option)	227
<code>--char_is_unsigned</code> (compiler option)	228
checksum	
calculation of	177
display format in C-SPY for symbol	181
<code>--checksum</code> (ielftool option)	431
cinttypes (DLIB header file)	378
ciso646 (library header file)	378
climits (DLIB header file)	378
clobber	132
locale (DLIB header file)	378
clock (DLIB library function), implementation-defined behavior in C89	471
clock (library function)	
implementation-defined behavior	458
clock.c	121
<code>__close</code> (DLIB library function)	117
<code>__CLREX</code> (intrinsic function)	346
clustering (compiler transformation)	200
disabling (<code>--no_clustering</code>)	243
<code>__CLZ</code> (intrinsic function)	346
cmain (system initialization code)	107
cmath (DLIB header file)	379
CMSIS integration	184

code	
ARM and Thumb, overview	57
interruption of execution	59
--code (ielfdump option)	433
code motion (compiler transformation)	199
disabling (--no_code_motion)	243
codeseg (pragma directive)	453, 467
__code, symbol used in library	381
command line options	
<i>See also</i> compiler options	
<i>See also</i> linker options	
part of compiler invocation syntax	209
part of linker invocation syntax	209
passing	210
typographic convention	26
command prompt icon, in this guide	27
.comment (ELF section)	406
comments	
after preprocessor directives	157
C++ style, using in C code	151
common block (call frame information)	146
common subexpr elimination (compiler transformation)	198
disabling (--no_cse)	244
Common.i (CFI header example file)	148
compilation date	
exact time of (__TIME__)	371
identifying (__DATE__)	369
compiler	
environment variables	211
invocation syntax	209
output from	212
compiler listing, generating (-l)	240
compiler object file	40
including debug information in (--debug, -r)	230
output from compiler	212
compiler optimization levels	196
compiler options	219
passing to compiler	210
reading from file (-f)	237
specifying parameters	221
summary	222
syntax	219
for creating skeleton code	138
instruction scheduling	200
--warnings_affect_exit_code	213
compiler platform, identifying	370
compiler transformations	195
compiler version number	371
compiling	
from the command line	47
syntax	209
complex numbers, supported in Embedded C++	162
complex (library header file)	377
complex.h (library header file)	375
compound literals	151
computer style, typographic convention	26
--config (linker option)	265
configuration	
basic project settings	48
__low_level_init	111
configuration file for linker. <i>See</i> linker configuration file	
configuration symbols	
for file input and output	117
for locale	118
for printf and scanf	115
in library configuration files	106, 112
in linker configuration files	400
specifying for linker	265
--config_def (linker option)	265
consistency, module	127
const	
declaring objects	301
non-top level	157
__constrange(), symbol used in library	381
__construction_by_bitwise_copy_allowed, symbol used in library	381
constseg (pragma directive)	453, 467
const_cast (cast operator)	162
contents, of this guide	22

control characters,
implementation-defined behavior 459
conventions, used in this guide 26
copyright notice 2
__CORE__ (predefined symbol). 369
core
 identifying 369
 selecting. 48
Cortex
 special considerations for interrupt functions. 64
 support for 228
cos (library function) 374
__cplusplus (predefined symbol) 369
--cpp_init_routine (linker option) 266
CPU
 specifying on command line for compiler 228
 specifying on command line for linker. 266
--cpu (compiler option). 228
--cpu (linker option) 266
CPU modes. 48
__CPU_MODE__ (predefined symbol) 369
--cpu_mode (compiler option) 229
--create (iarchive option). 433
csetjmp (DLIB header file) 379
csignal (DLIB header file) 379
cspy_support (pragma directive). 453, 467
CSTACK (ELF block). 407
 See also stack
 considering size of. 173
 setting up size for. 85
.cstart (ELF section) 407
cstartup (system startup code)
 customizing system initialization. 111
 source files for (DLIB). 107
cstdarg (DLIB header file) 379
cstdbool (DLIB header file) 379
cstddef (DLIB header file) 379
cstdio (DLIB header file) 379
cstdlib (DLIB header file). 379
cstring (DLIB header file). 379

ctgmath (library header file) 379
ctime (DLIB header file). 379
ctype.h (library header file). 375
cwctype.h (library header file) 379
C_INCLUDE (environment variable). 211
C-SPY
 debug support for C++ 168
 including debugging support 101
 interface to system termination 111
 Terminal I/O window, including debug support for . . . 103
C++
 See also Embedded C++ and Extended Embedded C++
 absolute location 193
 calling convention 138
 header files. 376
 language extensions. 170
 special function types. 65
 standard template library (STL). 377
 static member variables 193
 support for 33
--c++ (compiler option) 229
C++ header files 377
C++ terminology. 26
C++-style comments. 151
C89
 implementation-defined behavior. 461
 support for 151
--c89 (compiler option). 227

D

-D (compiler option). 230
-d (iarchive option) 434
data
 alignment of. 287
 different ways of storing 53
 located, declaring extern 193
 placing 191, 255, 405
 at absolute location 192

placing in registers	194
representation of	287
storage	53
data block (call frame information)	146
data pointers	296
data types	288
floating point	294
in C++	301
integer types	288
dataseg (pragma directive)	453, 467
data_alignment (pragma directive)	321
__data, symbol used in library	381
__DATE__ (predefined symbol)	369
date (library function), configuring support for	121
DC32 (assembler directive)	131
--debug (compiler option)	230
debug information, including in object file	230
.debug (ELF section)	406
DebugMon_Handler (exception function)	64
decimal point, implementation-defined behavior	459
declarations	
empty	158
in for loops	151
Kernighan & Ritchie	202
of functions	140
declarations and statements, mixing	151
declarators, implementation-defined behavior in C89	466
define block (linker directive)	389
define memory (linker directive)	384
define overlay (linker directive)	390
define region (linker directive)	385
define symbol (linker directive)	400
--define_symbol (linker option)	267
define_type_info (pragma directive)	453, 467
--delete (iarchive option)	434
delete (keyword)	55
denormalized numbers. <i>See</i> subnormal numbers	
--dependencies (compiler option)	231
--dependencies (linker option)	267
deque (STL header file)	377
destructors and interrupts, using	167
device description files, preconfigured for C-SPY	34
diagnostic messages	214
classifying as compilation errors	232
classifying as compilation remarks	232
classifying as compiler warnings	233
classifying as errors	245, 278
classifying as linker warnings	269
classifying as linking errors	268
classifying as linking remarks	268
disabling compiler warnings	250
disabling linker warnings	280
disabling wrapping of in compiler	250
disabling wrapping of in linker	281
enabling compiler remarks	254
enabling linker remarks	283
listing all used by compiler	233
listing all used by linker	269
suppressing in compiler	232
suppressing in linker	269
diagnostics	
iarchive	419
iobjmanip	424
isymexport	429
--diagnostics_tables (compiler option)	233
--diagnostics_tables (linker option)	269
diagnostics, implementation-defined behavior	445
diag_default (pragma directive)	321
--diag_error (compiler option)	232
--diag_error (linker option)	268
--no_fragments (compiler option)	245
--no_fragments (linker option)	278
diag_error (pragma directive)	322
--diag_remark (compiler option)	232
--diag_remark (linker option)	268
diag_remark (pragma directive)	322
--diag_suppress (compiler option)	232
--diag_suppress (linker option)	269

diag_suppress (pragma directive) 322
 --diag_warning (compiler option) 233
 --diag_warning (linker option) 269
 diag_warning (pragma directive) 323
 directives
 pragma 35, 317
 to the linker 383
 directory, specifying as parameter 220
 __disable_fiq (intrinsic function) 347
 __disable_interrupt (intrinsic function) 347
 __disable_irq (intrinsic function) 347
 --discard_unused_publics (compiler option) 233
 disclaimer 2
 DLIB 50, 375
 configurations 112
 configuring 94, 234
 documentation 24
 including debug support 101
 reference information. *See* the online help system . . . 373
 runtime environment 93
 --dlib_config (compiler option) 234
 DLib_Defaults.h (library configuration file) 106, 112
 __DLIB_FILE_DESCRIPTOR (configuration symbol) . . 117
 __DMB (intrinsic function) 347
 do not initialize (linker directive) 394
 document conventions 26
 documentation, overview of guides 23
 domain errors, implementation-defined behavior 454
 domain errors, implementation-defined behavior in C89
 (DLIB) 468
 __DOUBLE__ (predefined symbol) 369
 double (data type) 294
 identifying size of (__DOUBLE__) 369
 do_not_instantiate (pragma directive) 453, 467
 __DSB (intrinsic function) 348
 dynamic initialization 107
 and C++ 76
 dynamic memory 55
 dynamic RTTI data, including in the image 277

E

-e (compiler option) 235
 early_initialization (pragma directive) 453, 467
 --ec++ (compiler option) 235
 --edit (isymexport option) 434
 edition, of this guide 2
 --eec++ (compiler option) 235
 ELF utilities 417
 Embedded C++ 161
 differences from C++ 161
 enabling 235
 function linkage 140
 language extensions 161
 overview 161
 Embedded C++ Technical Committee 26
 embedded systems, IAR special support for 34
 __embedded_cplusplus (predefined symbol) 369
 empty region (in linker configuration file) 388
 __enable_fiq (intrinsic function) 348
 --enable_hardware_workaround (compiler option) 236
 --enable_hardware_workaround (linker option) 270
 __enable_interrupt (intrinsic function) 348
 __enable_irq (intrinsic function) 348
 --enable_multibytes (compiler option) 236
 --endian (compiler option) 236
 --entry (linker option) 270
 entry label, program 108
 enumerations, implementation-defined behavior 451
 enumerations, implementation-defined behavior in C89 . . 465
 enums
 data representation 289
 forward declarations of 156
 --enum_is_int (compiler option) 237
 environment
 implementation-defined behavior 446
 implementation-defined behavior in C89 461
 runtime (DLIB) 93
 environment names, implementation-defined behavior . . 447

environment variables	
C_INCLUDE	211
ILINKARM_CMD_LINE	211
QCCARM	211
environment (native),	
implementation-defined behavior	460
EQU (assembler directive)	253
ERANGE	454
ERANGE (C89)	468
errno value at underflow,	
implementation-defined behavior	457
errno.h (library header file)	375–376
error messages	216
classifying	245, 278
classifying for compiler	232
classifying for linker	268
range	90
error return codes	213
error (pragma directive)	323
--error_limit (compiler option)	237
--error_limit (linker option)	271
escape sequences, implementation-defined behavior ...	447
exception flags, for floating-point values	294
exception handling, missing from Embedded C++	161
exception stacks	174
exception (library header file)	377
exceptions, code for in section	408
--exception_tables (linker option)	271
exclude (stack usage control directive)	412
.exc.text (ELF section)	408
_Exit (library function)	110
exit (library function)	110
implementation-defined behavior	457
implementation-defined behavior in C89	471
_exit (library function)	110
__exit (library function)	110
export keyword, missing from Extended EC++	168
export (linker directive)	401
--export_builtin_config (linker option)	272
expressions (in linker configuration file)	401

extended command line file	
for compiler	237
for linker	272
passing options	210
Extended Embedded C++	162
enabling	235
extended keywords	303
enabling (-e)	235
overview	35
summary	306
syntax	
object attributes	305
type attributes on data objects	304
type attributes on data pointers	304
type attributes on functions	305
extended-selectors (in linker configuration file)	399
extern "C" linkage	167
--extract (iarchive option)	435
--extra_init (linker option)	272

F

-f (compiler option)	237
-f (IAR utility option)	435
-f (linker option)	272
fast interrupts	61
fatal error messages	216
fdopen, in stdio.h	380
fegettrapdisable	379
fegettrapenable	379
FENV_ACCESS, implementation-defined behavior	450
fenv.h (library header file)	376, 378
additional C functionality	379
fgetpos (library function), implementation-defined	
behavior	457
fgetpos (library function), implementation-defined	
behavior in C89	470
__FILE__ (predefined symbol)	369
file buffering, implementation-defined behavior	455

- file dependencies, tracking 231
- file paths, specifying for #include files 239
- file position, implementation-defined behavior 455
- file (zero-length), implementation-defined behavior 456
- filename
 - extension for device description files 34
 - extension for header files 34
 - of object executable image 281
 - of object file 251, 281
 - search procedure for 211
 - specifying as parameter 220
- filenames (legal), implementation-defined behavior 456
- fileno, in stdio.h 380
- files, implementation-defined behavior
 - handling of temporary 456
 - multibyte characters in 456
 - opening 456
- fill (lftool option) 435
- __fiq (extended keyword) 308
- FIQ_Handler (exception function) 60
- float (data type) 294
- floating-point constants
 - hexadecimal notation 151
 - hints 188
- floating-point environment, accessing or not 332
- floating-point expressions
 - contracting or not 333
 - using in preprocessor extensions 242
- floating-point format 294
 - hints 188
 - implementation-defined behavior 449
 - implementation-defined behavior in C89 464
 - special cases 295
 - 32-bits 294
 - 64-bits 295
- floating-point status flags 379
- floating-point unit 238
- float.h (library header file) 376
- FLT_EVAL_METHOD, implementation-defined behavior 449, 454, 458
- FLT_ROUND, implementation-defined behavior 449, 458
- fmod (library function), implementation-defined behavior in C89 469
- for loops, declarations in 151
- force_exceptions (linker option) 273
- force_output (linker option) 273
- formats
 - floating-point values 294
 - standard IEEE (floating point) 294
- fpu (compiler option) 238
- FP_CONTRACT, implementation-defined behavior 450
- fragmentation, of heap memory 56
- free (library function). *See also* heap 55
- fsetpos (library function), implementation-defined behavior 457
- fstream (library header file) 377
- ftell (library function), implementation-defined behavior . 457
- ftell (library function), implementation-defined behavior in C89 470
- Full DLIB (library configuration) 112
- __func__ (predefined symbol) 158, 370
- __FUNCTION__ (predefined symbol) 158, 370
- function calls
 - calling convention 139
 - stack image after 143
- function declarations, Kernighan & Ritchie 202
- function execution, in RAM 58
- function inlining (compiler transformation) 198
 - disabling (--no_inline) 245
- function pointers 296
- function prototypes 201
 - enforcing 254
- function (pragma directive) 453, 467
- function (stack usage control directive) 411
- functional (STL header file) 377
- functions 57
 - C++ and special function types 65
 - declaring 140, 201
 - inlining 151, 198, 201, 324

interrupt	59
intrinsic	129
parameters	141
placing in memory	191, 193, 255
recursive	
avoiding	201
storing data on stack	54
reentrancy (DLIB)	374
related extensions	57
return values from	143
special function types	59
function_effects (pragma directive)	453, 467
function-spec (in stack usage control file)	413

G

getenv (library function), configuring support for	120
getw, in stdio.h	380
getzone (library function), configuring support for	121
getzone.c	121
__get_BASEPRI (intrinsic function)	348
__get_CONTROL (intrinsic function)	349
__get_CPSR (intrinsic function)	349
__get_FAULTMASK (intrinsic function)	349
__get_FPSCR (intrinsic function)	349
__get_interrupt_state (intrinsic function)	349
__get_IPSR (intrinsic function)	350
__get_LR (intrinsic function)	350
__get_MSP (intrinsic function)	350
__get_PRIMASK (intrinsic function)	351
__get_PSP (intrinsic function)	351
__get_PSR (intrinsic function)	351
__get_SB (intrinsic function)	351
__get_SP (intrinsic function)	351
GRP_COMDAT, group type	425
--guard_calls (compiler option)	239
guidelines, reading	21

H

Harbison, Samuel P.	25
HardFault_Handler (exception function)	64
hardware support in compiler	93
hash_map (STL header file)	378
hash_set (STL header file)	378
__has_constructor, symbol used in library	381
__has_destructor, symbol used in library	381
hdrstop (pragma directive)	453, 467
header files	
C	375
C++	376–377
library	373
special function registers	203
STL	377
DLib_Defaults.h	106, 112
including stdbool.h for bool	289
including stddef.h for wchar_t	290
header names, implementation-defined behavior	452
--header_context (compiler option)	239
heap	
dynamic memory	55
storing data	53
heap size	
and standard I/O	175
changing default	86
HEAP (ELF section)	175, 408
heap (zero-sized), implementation-defined behavior	457
hide (isymexport directive)	428
hints	
for good code generation	200
implementation-defined behavior	450
using efficient data types	187
 -I (compiler option)	239
IAR Command Line Build Utility	106

- IAR Systems Technical Support 216
- iarbuild.exe (utility) 106
- iarchive 417
 - options summary 419
- __iar_cos_accuratef (library function) 374
- __iar_cos_accuratel (library function) 374
- __IAR_DLIB_PERTHREAD_INIT_SIZE (macro) 126
- __IAR_DLIB_PERTHREAD_SIZE (macro) 125
- __IAR_DLIB_PERTHREAD_SYMBOL_OFFSET (symbolptr) 126
- __iar_maximum_atexit_calls 86
- __iar_pow_accuratef (library function) 374
- __iar_pow_accuratel (library function) 374
- __iar_program_start (label) 108
- __iar_ReportAssert (library function) 122
- __iar_sin_accuratef (library function) 374
- __iar_sin_accuratel (library function) 374
- __IAR_SYSTEMS_ICC__ (predefined symbol) 370
- __iar_tan_accuratef (library function) 374
- __iar_tan_accuratel (library function) 374
- iar.debug (ELF section) 406
- iar.dynexit (ELF section) 408
- IA64 ABI 182
 - __ICCARM__ (predefined symbol) 370
- icons, in this guide 27
- IDE
 - building a library from 106
 - overview of build tools 31
- identifiers, implementation-defined behavior 447
- identifiers, implementation-defined behavior in C89 462
- IEEE format, floating-point values 294
- ifldump 422
 - options summary 423
- iflftool 420
 - options summary 421
- if (linker directive) 403
- ihex (iflftool option) 436
- ILINK options. *See* linker options
- ILINKARM_CMD_LINE (environment variable) 211
- ILINK. *See* linker
- image_input (linker option) 274
- implementation-defined behavior
 - C89 461
 - Standard C 445
- important_typedef (pragma directive) 453, 467
- include files
 - including before source files 252
 - specifying 211
- include (linker directive) 403
- include_alias (pragma directive) 323
- infinity 295
- infinity (style for printing), implementation-defined behavior 456
- inheritance, in Embedded C++ 161
- initialization
 - changing default 86
 - C++ dynamic 76
 - dynamic 107
 - manual 87
 - packing algorithm for 86
 - single-value 158
- initialize (linker directive) 392
- initializers, static 157
- .init_array (section) 408
- inline (linker option) 274
- inline assembler 131
 - See also* assembler language interface
- inline functions 151
 - in compiler 198
- inline (pragma directive) 324
- inlining functions, implementation-defined behavior 451
- installation directory 26
- instantiate (pragma directive) 453, 467
- instruction scheduling (compiler option) 200
- int (data type) signed and unsigned 289
- integer types 288
 - casting 296
 - implementation-defined behavior 449
- intptr_t 297
- ptrdiff_t 297

size_t	296
uintptr_t	297
integers, implementation-defined behavior in C89	463
integral promotion	202
Intel hex	173
Intel IA64 ABI	182
internal error	216
interrupt functions	59
fast interrupts	61
in ARM Cortex	64
nested interrupts	61
operations	63
software interrupts	62
interrupt handler. <i>See</i> interrupt service routine	
interrupt service routine	59
interrupt state, restoring	358
interrupt vector table	64
.intvec section	408
interrupts	
processor state	54
using with EC++ destructors	167
__interwork (extended keyword)	308
--interwork (compiler option)	239
interworking code	49
intptr_t (integer type)	297
__intrinsic (extended keyword)	308
intrinsic functions	
for Neon	345
overview	129
summary	337
intrinsics.h (header file)	337
inttypes.h (library header file)	376
.intvec (section)	408
invocation syntax	209
ioobjmanip	423
options summary	424
ioomanip (library header file)	377
ios (library header file)	377
iosfwd (library header file)	377

iostream (library header file)	377
__irq (extended keyword)	309
IRQ_Handler (exception function)	60
IRQ_STACK (section)	409
__ISB (intrinsic function)	352
iso646.h (library header file)	376
istream (library header file)	377
ismlexport	426
options summary	427
italic style, in this guide	26
iterator (STL header file)	378
I/O register. <i>See</i> SFR	

J

Josuttis, Nicolai M.	25
----------------------	----

K

--keep (linker option)	275
keep (linker directive)	394
keep_definition (pragma directive)	453, 468
Kernighan & Ritchie function declarations	202
disallowing	254
Kernighan, Brian W.	25
keywords	303
extended, overview of	35

L

-l (compiler option)	240
for creating skeleton code	138
labels	157
assembler, making public	253
__iar_program_start	108
__program_start	108
Labrosse, Jean J.	25
Lajoie, Josée	25

- language extensions
 - Embedded C++ 161
 - enabling using pragma 325
 - enabling (-e). 235
- language overview 33
- language (pragma directive) 325
- __LDC (intrinsic function) 352
- __LDCL (intrinsic function) 352
- __LDCL_noidx (intrinsic function) 352
- __LDC_noidx (intrinsic function) 352
- __LDC2 (intrinsic function) 352
- __LDC2L (intrinsic function) 352
- __LDC2L_noidx (intrinsic function) 352
- __LDC2_noidx (intrinsic function) 352
- __LDREX (intrinsic function) 353
- __LDREXB (intrinsic function) 353
- __LDREXD (intrinsic function) 353
- __LDREXH (intrinsic function) 353
- legacy (compiler option). 241
- Lempel-Ziv-Welch algorithm, for packing initializers . . . 392
- libraries
 - reason for using 40
 - standard template library 377
 - using a prebuilt 95
- library configuration files
 - DLIB 112
 - DLib_Defaults.h 106, 112
 - modifying 106
 - specifying 234
- library documentation 373
- library features, missing from Embedded C++ 162
- library files, linker search path to (--search) 283
- library functions 373
 - summary, DLIB 375
- library header files 373
- library modules
 - introduction 68
 - overriding 105
- library object files 374
- library options, setting 51
- library project template 50
 - using 106
- library_default_requirements (pragma directive) . . . 453, 468
- library_provides (pragma directive) 453, 468
- library_requirement_override (pragma directive) . . . 453, 468
- lightbulb icon, in this guide. 27
- limits (library header file) 377
- limits.h (library header file) 376
- __LINE__ (predefined symbol) 370
- linkage, C and C++ 140
- linker. 67
 - output from 214
- linker configuration file
 - for placing code and data 71
 - in depth 383, 411
 - overview of 383, 411
 - selecting. 81
- linker object executable image
- specifying filename of (-o) 281
- linker options 261
 - reading from file (-f) 272
 - summary 261
 - typographic convention 26
- linking
 - from the command line 47
 - in the build process 40
 - introduction 67
 - process for 69
- Lippman, Stanley B. 25
- list (STL header file). 378
- listing, generating 240
- literals, compound. 151
- literature, recommended 24
- __LITTLE_ENDIAN__ (predefined symbol). 370
- __little_endian (extended keyword) 309
- little-endian (byte order) 49
- local symbols, removing from ELF image 279
- local variables, *See* auto variables

locale	
adding support for in library	119
changing at runtime	119
implementation-defined behavior	448, 459
library header file	377
removing support for	119
support for	118
locale.h (library header file)	376
located data, declaring extern	193
location (pragma directive)	192, 326
--lock_regs (compiler option)	241
--log (linker option)	275
--log_file (linker option)	276
long double (data type)	294
long float (data type), synonym for double	157
longjmp, restrictions for using	375
loop unrolling (compiler transformation)	198
disabling	249
loop-invariant expressions	199
__low_level_init	108
customizing	111
initialization phase	43
low_level_init.c	107
low_level_init.s	107
low-level processor operations	152, 337
accessing	129
__lseek (library function)	117
lzw, packing algorithm for initializers	392

M

macros	
embedded in #pragma optimize	328
ERANGE (in errno.h)	454, 468
inclusion of assert	371
NULL	
implementation-defined behavior in C89	468
NULL, implementation-defined behavior	455
substituted in #pragma directives	152

variadic	151
--macro_positions_in_diagnostics (compiler option)	242
main (function)	
definition (C89)	461
implementation-defined behavior	446
malloc (library function)	
<i>See also</i> heap	55
implementation-defined behavior in C89	470
--mangled_names_in_messages (linker option)	276
Mann, Bernhard	25
-map (linker option)	276
map file, producing	276
map (STL header file)	378
math functions rounding mode,	
implementation-defined behavior	458
math.h (library header file)	376
MB_LEN_MAX, implementation-defined behavior	458
__MCR (intrinsic function)	353
MemManage_Handler (exception function)	64
memory	
allocating in C++	55
dynamic	55
heap	55
non-initialized	205
RAM, saving	201
releasing in C++	55
stack	54
saving	201
used by global or static variables	53
memory clobber	132
memory layout, ARM	53
memory management, type-safe	161
memory map, output from linker	214
memory (pragma directive)	453, 468
memory (STL header file)	378
__memory_of, symbol used in library	381
-merge_duplicate_sections (linker option)	277
message (pragma directive)	327
messages	
disabling	256, 284

forcing 327
 Meyers, Scott 25
 --mfc (compiler option) 242
 --migration_preprocessor_extensions (compiler option) 242
 migration, from earlier IAR compilers 24
 MISRA C, documentation 24
 --misrac (compiler option) 223
 --misrac_verbose (compiler option) 223
 --misrac_verbose (linker option) 262
 --misrac1998 (compiler option) 223
 --misrac1998 (linker option) 262
 --misrac2004 (compiler option) 223
 --misrac2004 (linker option) 262
 mode changing, implementation-defined behavior 456
 module consistency 127
 rtmodel 330
 modules, introduction 68
 module_name (pragma directive) 453, 468
 module-spec (in stack usage control file) 414
 Motorola S-records 173
 __MRC (intrinsic function) 354
 multibyte character support 236
 multibyte characters, implementation-defined
 behavior 447, 459
 multiple inheritance
 in Extended EC++ 162
 missing from Embedded C++ 161
 missing from STL 162
 multi-file compilation 196
 mutable attribute, in Extended EC++ 162, 169

N

name (in stack usage control file) 414
 names block (call frame information) 146
 namespace support
 in Extended EC++ 162, 169
 missing from Embedded C++ 162
 naming conventions 27

NaN
 implementation of 295
 implementation-defined behavior 456
 native environment,
 implementation-defined behavior 460
 NDEBUG (preprocessor symbol) 371
 Neon intrinsic functions 345
 __nested (extended keyword) 309
 nested interrupts 61
 new (keyword) 55
 new (library header file) 377
 NMI_Handler (exception function) 64
 non-initialized variables, hints for 205
 non-scalar parameters, avoiding 201
 NOP (assembler instruction) 354
 __noreturn (extended keyword) 310
 Normal DLIB (library configuration) 112
 Not a number (NaN) 295
 --no_clustering (compiler option) 243
 --no_code_motion (compiler option) 243
 --no_const_align (compiler option) 243
 --no_cse (compiler option) 244
 --no_dynamic_rtti_elimination (linker option) 277
 --no_exceptions (compiler option) 244
 --no_exceptions (linker option) 278
 __no_init (extended keyword) 205, 310
 --no_inline (compiler option) 245
 --no_library_search (linker option) 278
 --no_locals (linker option) 279
 --no_loop_align (compiler option) 245
 --no_mem_idioms (compiler option) 246
 __no_operation (intrinsic function) 354
 --no_path_in_file_macros (compiler option) 246
 no_pch (pragma directive) 453, 468
 --no_range_reservations (linker option) 279
 --no_remove (linker option) 279
 --no_rtti (compiler option) 246
 --no_rw_dynamic_init (compiler option) 247
 --no_scheduling (compiler option) 247
 --no_static_destruction (compiler option) 247

--no_strtab (ielfdump option)	436
--no_system_include (compiler option)	248
--no_tbaa (compiler option)	248
--no_typedefs_in_diagnostics (compiler option).	248
--no_unaligned_access (compiler option)	249
--no_unroll (compiler option)	249
--no_veneer (linker option)	280
--no_vfe (compiler option)	280
--no_warnings (compiler option)	250
--no_warnings (linker option)	280
--no_wrap_diagnostics (compiler option)	250
--no_wrap_diagnostics (linker option)	281
NULL	
implementation-defined behavior.	455
implementation-defined behavior in C89.	468
pointer constant, relaxation to Standard C	156
numbers (in linker configuration file)	402
numeric conversion functions,	
implementation-defined behavior	460
numeric (STL header file).	378

O

-O (compiler option)	250
-o (compiler option)	251
-o (iarchive option)	437
-o (ielfdump option)	437
-o (linker option).	281
object attributes.	305
object filename, specifying (-o)	251, 281
object files, linker search path to (--search).	283
object_attribute (pragma directive)	205, 327
once (pragma directive)	453, 468
--only_stdout (compiler option)	251
--only_stdout (linker option).	281
__open (library function)	117
operators	
<i>See also</i> @ (operator)	
for cast, in Extended EC++	162

for cast, missing from Embedded C++.	162
for region expressions	387
for section control	155
precision for 32-bit float	294
precision for 64-bit float	295
sizeof, implementation-defined behavior	459
variants for cast	169
_Pragma (preprocessor)	151
__ALIGNOF__, for alignment control.	154
?, language extensions for	170
optimization	
clustering, disabling.	243
code motion, disabling	243
common sub-expression elimination, disabling	244
configuration	49
disabling	197
function inlining, disabling (--no_inline)	245
hints	200
loop unrolling, disabling	249
scheduling, disabling	247
specifying (-O).	250
techniques	197
type-based alias analysis, disabling (--tbaa).	248
using inline assembler code	132
using pragma directive	327
optimization levels	196
optimize (pragma directive)	327
option parameters	219
options, compiler. <i>See</i> compiler options	
options, iarchive. <i>See</i> iarchive options	
options, ielfdump. <i>See</i> ielfdump options	
options, ielftool. <i>See</i> ielftool options	
options, iobjmanip. <i>See</i> iobjmanip options	
options, isymexport. <i>See</i> isymexport options	
options, linker. <i>See</i> linker options	
--option_name (compiler option)	270
Oram, Andy	25
ostream (library header file)	377

output
 from preprocessor 252
 specifying for linker 47
 --output (compiler option). 251
 --output (iarchive option) 437
 --output (ielfdump option) 437
 --output (linker option) 281
 overhead, reducing 198

P

pack (pragma directive) 298, 328
 packbits, packing algorithm for initializers 392
 __packed (extended keyword). 310
 packed structure types. 298
 packing, algorithms for initializers 392
 parameters
 function 141
 hidden 142
 non-scalar, avoiding 201
 register 141–142
 rules for specifying a file or directory 220
 specifying 221
 stack. 141–142
 typographic convention 26
 part number, of this guide 2
 __pcrel (extended keyword) 306
 PendSV_Handler (exception function) 64
 permanent registers 141
 perror (library function),
 implementation-defined behavior in C89 470
 --pi_veneer (linker option) 281
 __PKHBT (intrinsic function). 354
 __PKHTB (intrinsic function). 355
 place at (linker directive) 395
 place in (linker directive) 396
 placement
 code and data 405
 in named sections. 193

 of code and data, introduction to 71
 --place_holder (linker option). 282
 plain char, implementation-defined behavior 447
 pointer types 296
 mixing 157
 pointers
 casting 296
 data 296
 function 296
 implementation-defined behavior. 450
 implementation-defined behavior in C89 465
 polymorphic RTTI data, including in the image 277
 polymorphism, in Embedded C++ 161
 porting, code containing pragma directives. 318
 possible calls (stack usage control directive). 412
 pow (library function). 121
 alternative implementation of. 374
 powXp (library function) 121
 pragma directives 35
 summary 317
 for absolute located data 192
 list of all recognized. 452
 list of all recognized (C89). 467
 pack 298, 328
 _Pragma (preprocessor operator) 151
 predefined symbols
 overview 35
 summary 368
 --predef_macro (compiler option). 251
 Prefetch_Handler (exception function) 60
 --preinclude (compiler option) 252
 .preinit_array (section) 409
 .prepreinit_array (section). 409
 --preprocess (compiler option) 252
 preprocessor
 operator (_Pragma) 151
 output. 252
 overview of 367

preprocessor directives	
comments at the end of	157
implementation-defined behavior.	452
implementation-defined behavior in C89	466
#pragma	317
preprocessor extensions	
compatibility	242
__VA_ARGS__	151
#warning message	372
preprocessor symbols	368
defining	230, 267
preserved registers	141
__PRETTY_FUNCTION__ (predefined symbol)	370
primitives, for special functions	59
print formatter, selecting	100
printf (library function)	99
choosing formatter	99
configuration symbols	115
implementation-defined behavior.	457
implementation-defined behavior in C89	470
__printf_args (pragma directive)	329
printing characters, implementation-defined behavior	459
processor configuration	48
processor operations	
accessing	129
low-level	152, 337
program entry label	108
program termination, implementation-defined behavior	446
programming hints	200
__program_start (label)	108
projects	
basic settings for	48
setting up for a library	106
prototypes, enforcing	254
ptrdiff_t (integer type)	297
PUBLIC (assembler directive)	253
publication date, of this guide	2
--public_equ (compiler option)	253
public_equ (pragma directive)	453, 468

putenv (library function), absent from DLIB	120
putw, in stdio.h	380

Q

__QADD (intrinsic function)	355
__QADD8 (intrinsic function)	355
__QADD16 (intrinsic function)	355
__QASX (intrinsic function)	355
QCCARM (environment variable)	211
__QCFI (intrinsic function)	355
__QDADD (intrinsic function)	355
__QDOUBLE (intrinsic function)	356
__QDSUB (intrinsic function)	355
__QFlag (intrinsic function)	356
__QSAX (intrinsic function)	355
__QSUB (intrinsic function)	355
__QSUB16 (intrinsic function)	355
__QSUB8 (intrinsic function)	355
qualifiers	
const and volatile	299
implementation-defined behavior	451
implementation-defined behavior in C89	466
queue (STL header file)	378

R

-r (compiler option)	230
-r (iarchive option)	439
raise (library function), configuring support for	121
raise.c	121
RAM	
example of declaring region	72
execution	58
initializers copied from ROM	46
running code from	88
saving memory	201
__ramfunc (extended keyword)	58, 311
--ram_reserve_ranges (isymexport option)	437

- range errors 90
- raw (ielfdump option) 438
- __RBIT (intrinsic function) 356
- __read (library function) 117
 - customizing 113
- read formatter, selecting 101
- reading guidelines 21
- reading, recommended 24
- realloc (library function) 55
 - implementation-defined behavior in C89 470
 - See also* heap
- recursive functions
 - avoiding 201
 - storing data on stack 54
- redirect (linker option) 282
- reentrancy (DLIB) 374
- reference information, typographic convention 26
- region expression (in linker configuration file) 387
- region literal (in linker configuration file) 385
- register keyword, implementation-defined behavior 450
- register parameters 141–142
- registered trademarks 2
- registers
 - assigning to parameters 142
 - callee-save, stored on stack 54
 - implementation-defined behavior in C89 465
 - in assembler-level routines 139
 - preserved 141
 - scratch 141
- reinterpret_cast (cast operator) 162
- .rel (ELF section) 406
- .rela (ELF section) 406
- relaxed_fp (compiler option) 253
- relay, *see* veneers 90
- relocation errors, resolving 90
- remark (diagnostic message) 215
 - classifying for compiler 232
 - classifying for linker 268
 - enabling in compiler 254
 - enabling in linker 283
- remarks (compiler option) 254
- remarks (linker option) 283
- remove (library function) 117
 - implementation-defined behavior 456
 - implementation-defined behavior in C89 (DLIB) 470
- remove_section (iobjmanip option) 438
- remquo, magnitude of 455
- rename (isymexport directive) 428
- rename (library function) 117
 - implementation-defined behavior 456
 - implementation-defined behavior in C89 (DLIB) 470
- rename_section (iobjmanip option) 439
- rename_symbol (iobjmanip option) 439
- replace (iarchive option) 439
- required (pragma directive) 329
- require_prototypes (compiler option) 254
- reserve_ranges (isymexport option) 440
- __reset_QC_flag (intrinsic function) 356
- __reset_Q_flag (intrinsic function) 356
- return values, from functions 143
- __REV (intrinsic function) 357
- __REVSH (intrinsic function) 357
- __REV16 (intrinsic function) 357
- Ritchie, Dennis M. 25
- ROM to RAM, copying 88
- __root (extended keyword) 311
- __ROPI__ (predefined symbol) 370
- ropi (compiler option) 254
- routines, time-critical 129, 152, 337
- rtmodel (assembler directive) 128
- rtmodel (pragma directive) 330
- RTTI data (dynamic), including in the image 277
- rtti support, missing from STL 162
- runtime environment
 - DLIB 93
 - setting options for 51
 - setting up (DLIB) 94

runtime libraries (DLIB)	
introduction	373
customizing system startup code	111
customizing without rebuilding	98
filename syntax	96
overriding modules in	105
using prebuilt	95
runtime library	
setting up from command line	51
setting up from IDE	50
runtime model attributes	127
runtime model definitions	330
runtime type information, missing from Embedded C++	162
--rwpi (compiler option)	255

S

-S (iarchive option)	441
-s (ielfdump option)	440
__SADD8 (intrinsic function)	357
__SADD16 (intrinsic function)	357
__SASX (intrinsic function)	357
__sbrel (extended keyword)	306
scanf (library function)	
configuration symbols	115
implementation-defined behavior	457
implementation-defined behavior in C89 (DLIB)	470
scanf (library function), choosing formatter (DLIB)	100
__scanf_args (pragma directive)	331
scheduling (compiler transformation)	200
disabling	247
scratch registers	141
--search (linker option)	283
search path to library files (--search)	283
search path to object files (--search)	283
--section (ielfdump option)	440
--section (compiler option)	255
section (pragma directive)	331

sections	405
summary	405
allocation of	71
declaring (#pragma section)	331
introduction	68
specifying (--section)	255
__section_begin (extended operator)	155
__section_end (extended operator)	155
__section_size (extended operator)	155
section-selectors (in linker configuration file)	397
__SEL (intrinsic function)	357
--self_reloc (ielftool option)	441
--semihosting (linker option)	283
semihosting, overview	102
--separate_cluster_for_initialized_variables (compiler option)	256
set (STL header file)	378
setjmp.h (library header file)	376
setlocale (library function)	119
settings, basic for project configuration	48
__set_BASEPRI (intrinsic function)	357
__set_CONTROL (intrinsic function)	357
__set_CPSR (intrinsic function)	358
__set_FAULTMASK (intrinsic function)	358
__set_FPSCR (intrinsic function)	358
__set_interrupt_state (intrinsic function)	358
__set_LR (intrinsic function)	358
__set_MSP (intrinsic function)	359
__set_PRIMASK (intrinsic function)	359
__set_PSP (intrinsic function)	359
__set_SB (intrinsic function)	359
__set_SP (intrinsic function)	359
__SEV (intrinsic function)	359
severity level, of diagnostic messages	215
specifying	216
SFR	
accessing special function registers	203
declaring extern special function registers	193
__SHADD8 (intrinsic function)	360
__SHADD16 (intrinsic function)	360

- shared object 213, 277
- __SHASX (intrinsic function). 360
- short (data type) 289
- show (ismexport directive) 428
- __SHSAX (intrinsic function). 360
- .shstrtab (ELF section) 406
- __SHSUB16 (intrinsic function). 360
- __SHSUB8 (intrinsic function). 360
- signal (library function)
 - configuring support for 121
 - implementation-defined behavior. 455
 - implementation-defined behavior in C89. 469
- signals, implementation-defined behavior. 446
 - at system startup 446
- signal.c 121
- signal.h (library header file) 376
- signed char (data type) 288–289
 - specifying 227
- signed int (data type). 289
- signed long long (data type) 289
- signed long (data type) 289
- signed short (data type). 289
- silent (compiler option) 256
- silent (iarchive option) 441
- silent (ielftool option). 441
- silent (linker option). 284
- silent operation
 - specifying in compiler 256
 - specifying in linker 284
- simple (ielftool option). 442
- sin (library function). 374
- 64-bits (floating-point format) 295
- sizeof, using in preprocessor extensions 242
- size_t (integer type) 296
- skeleton code, creating for assembler language interface . 137
- skeleton.s (assembler source output). 138
- skip_dynamic_initialization (linker option) 284
- slist (STL header file) 378
- smallest, packing algorithm for initializers 392
- __SMLABB (intrinsic function). 360
- __SMLABT (intrinsic function). 360
- __SMLAD (intrinsic function) 360
- __SMLADX (intrinsic function). 360
- __SMLALBB (intrinsic function). 360
- __SMLALBT (intrinsic function). 360
- __SMLALD (intrinsic function). 360
- __SMLALDX (intrinsic function) 360
- __SMLALTB (intrinsic function) 360
- __SMLALTT (intrinsic function) 360
- __SMLATB (intrinsic function) 360
- __SMLATT (intrinsic function) 360
- __SMLAWB (intrinsic function). 360
- __SMLAWT (intrinsic function). 360
- __SMLSD (intrinsic function) 360
- __SMLS DX (intrinsic function) 360
- __SMLS LD (intrinsic function) 360
- __SMLS LD X (intrinsic function). 360
- __SMMLA (intrinsic function). 361
- __SMMLAR (intrinsic function) 361
- __SMMLS (intrinsic function) 361
- __SMMLSR (intrinsic function). 361
- __SMMUL (intrinsic function). 361
- __SMMULR (intrinsic function) 361
- __SMUAD (intrinsic function) 361
- __SMUL (intrinsic function) 361
- __SMULBB (intrinsic function). 361
- __SMULBT (intrinsic function) 361
- __SMULTB (intrinsic function) 361
- __SMULTT (intrinsic function) 361
- __SMULWB (intrinsic function) 361
- __SMULWT (intrinsic function). 361
- __SMUSD (intrinsic function) 361
- __SMUSD X (intrinsic function). 361
- software interrupts 62
- source files, list all referred. 239
- space characters, implementation-defined behavior 455
- special function registers (SFR) 203

special function types	59
overview	35
sprintf (library function)	99
choosing formatter	99
--srec (ielftool option)	442
--srec-len (ielftool option)	442
--srec-s3only (ielftool option)	443
__SSAT (intrinsic function)	362
__SSAT16 (intrinsic function)	362
__SSAX (intrinsic function)	357
sscanf (library function), choosing formatter (DLIB)	100
sstream (library header file)	377
__SSUB16 (intrinsic function)	357
__SSUB8 (intrinsic function)	357
stack	54
advantages and problems using	54
block for holding	407
cleaning after function return	144
contents of	54
exception	174
layout	142
saving space	201
setting up size for	85
size	173
stack parameters	141–142
stack pointer	54
stack (STL header file)	378
__stackless (extended keyword)	312
--stack_usage_control (compiler option)	284
stack-size (in stack usage control file)	415
Standard C	
library compliance with	373
specifying strict usage	256
standard error	
redirecting in compiler	251
redirecting in linker	281
standard input	113
standard output	113
specifying in compiler	251
specifying in linker	281
standard template library (STL)	
in C++	377
in Extended EC++	162, 169
missing from Embedded C++	162
startup system. <i>See</i> system startup	
statements, implementation-defined behavior in C89	466
static clustering (compiler transformation)	200
static variables	53
taking the address of	201
static_assert()	154
static_cast (cast operator)	162
status flags for floating-point	379
__STC (intrinsic function)	362
__STCL (intrinsic function)	362
__STCL_nidx (intrinsic function)	363
__STC_nidx (intrinsic function)	363
__STC2 (intrinsic function)	362
__STC2L (intrinsic function)	362
__STC2L_nidx (intrinsic function)	363
__STC2_nidx (intrinsic function)	363
std namespace, missing from EC++	
and Extended EC++	169
stdarg.h (library header file)	376
stdbool.h (library header file)	289, 376
__STDC__ (predefined symbol)	371
STDC CX_LIMITED_RANGE (pragma directive)	332
STDC FENV_ACCESS (pragma directive)	332
STDC FP_CONTRACT (pragma directive)	333
__STDC_VERSION__ (predefined symbol)	371
stddef.h (library header file)	290, 376
stderr	117, 251, 281
stdexcept (library header file)	377
stdin	117
implementation-defined behavior in C89 (DLIB)	469
stdint.h (library header file)	376, 379
stdio.h (library header file)	376
stdio.h, additional C functionality	380
stdlib.h (library header file)	376

- stdout 117, 251, 281
 - implementation-defined behavior. 455
 - implementation-defined behavior in C89 (DLIB) 469
- Steele, Guy L. 25
- steering file, input to isymexport. 427
- STL. 169
- strcasecmp, in string.h 380
- strdup, in string.h 380
- streambuf (library header file). 377
- streams
 - implementation-defined behavior. 446
 - supported in Embedded C++ 162
- strerror (library function), implementation-defined behavior 460
- strerror (library function), implementation-defined behavior in C89 (DLIB) 471
- __STREX (intrinsic function). 363
- __STREXB (intrinsic function) 363
- __STREXD (intrinsic function) 363
- __STREXH (intrinsic function) 363
- strict (compiler option). 256
- string (library header file) 377
- strings, supported in Embedded C++ 162
- string.h (library header file) 376
- string.h, additional C functionality 380
- strip (ielftool option) 443
- strip (iobjmanip option) 443
- strip (linker option) 285
- strncasecmp, in string.h. 380
- strlen, in string.h 380
- Stroustrup, Bjarne 25
- strstream (library header file) 377
- .strtab (ELF section) 406
- structure types
 - alignment 297–298
 - layout of. 297
 - packed 298
- structures
 - aligning 329
 - anonymous. 154, 189
 - implementation-defined behavior. 451
 - implementation-defined behavior in C89 465
 - packing and unpacking 189
- subnormal numbers. 294–295
- support, technical 216
- Sutter, Herb. 25
- SVC #immed, for software interrupts 62
- SVC_Handler (exception function). 64
- __swi (extended keyword) 312
- SWI_Handler (exception function). 60
- swi_number (pragma directive) 333
- SWO, directing stdout/stderr via. 101
- __SWP (intrinsic function) 364
- __SWPB (intrinsic function). 364
- __SXTAB (intrinsic function). 364
- __SXTAB16 (intrinsic function). 364
- __SXTAH (intrinsic function). 364
- __SXTB16 (intrinsic function) 364
- symbol names, using in preprocessor extensions 242
- symbols
 - anonymous, creating 151
 - directing from one to another. 282
 - including in output. 330
 - local, removing from ELF image 279
 - overview of predefined. 35
 - preprocessor, defining 230, 267
- symbols (iarchive option). 443
- .symtab (ELF section). 406
- syntax
 - command line options 219
 - extended keywords. 304–305
 - invoking compiler and linker 209
- system function, implementation-defined behavior. . 447, 457
- system startup
 - customizing 111
 - DLIB 107
 - initialization phase. 43
- system termination
 - C-SPY interface to. 111

DLIB	110
system (library function)	
configuring support for	120
implementation-defined behavior in C89 (DLIB)	471
system_include (pragma directive)	453, 468
--system_include_dir (compiler option)	257
SysTick_Handler (exception function)	64

T

-t (iarchive option)	444
tan (library function)	374
__task (extended keyword)	313
technical support, IAR Systems	216
template support	
in Extended EC++	162, 168
missing from Embedded C++	161
Terminal I/O window	
making available (DLIB)	103
not supported when	105
terminal I/O, debugger runtime interface for	102
terminal output, speeding up	103
termination of system. <i>See</i> system termination	
termination status, implementation-defined behavior	457
terminology	26
tgmath.h (library header file)	376
32-bits (floating-point format)	294
this (pointer)	138
__thumb (extended keyword)	314
--thumb (compiler option)	257
Thumb, CPU mode	48
__TIME__ (predefined symbol)	371
time zone (library function)	
implementation-defined behavior in C89	471
time zone (library function), implementation-defined behavior	458
time-critical routines	129, 152, 337
time.c	121

time.h (library header file)	376
additional C functionality	380
time32 (library function), configuring support for	121
time64 (library function), configuring support for	121
tips, programming	200
--toc (iarchive option)	444
tools icon, in this guide	26
trademarks	2
transformations, compiler	195
translation, implementation-defined behavior	445
translation, implementation-defined behavior in C89	461
type attributes	303
specifying	334
type qualifiers	
const and volatile	299
implementation-defined behavior	451
implementation-defined behavior in C89	466
typedefs	
excluding from diagnostics	248
repeated	157
using in preprocessor extensions	242
typeinfo (library header file)	377
type_attribute (pragma directive)	333
type-based alias analysis (compiler transformation)	199
disabling	248
type-safe memory management	161
typographic conventions	26

U

__UADD8 (intrinsic function)	364
__UADD16 (intrinsic function)	364
__UASX (intrinsic function)	364
uchar.h (library header file)	376
__UHADD8 (intrinsic function)	364
__UHADD16 (intrinsic function)	364
__UHASX (intrinsic function)	364
__UHSAX (intrinsic function)	364
__UHSUB16 (intrinsic function)	364

__UHSUB8 (intrinsic function) 364
 uintptr_t (integer type) 297
 __UMAAL (intrinsic function) 364
 Undefined_Handler (exception function) 60
 underflow errors, implementation-defined behavior 454
 underflow range errors,
 implementation-defined behavior in C89 468
 __ungetchar, in stdio.h 380
 unions
 anonymous. 154, 189
 implementation-defined behavior. 451
 implementation-defined behavior in C89. 465
 universal character names, implementation-defined
 behavior 452
 unsigned char (data type) 288–289
 changing to signed char 227
 unsigned int (data type) 289
 unsigned long long (data type) 289
 unsigned long (data type) 289
 unsigned short (data type) 289
 __UQADD8 (intrinsic function) 365
 __UQADD16 (intrinsic function) 365
 __UQASX (intrinsic function) 365
 __UQSAX (intrinsic function) 365–366
 __UQSUB16 (intrinsic function) 365–366
 __UQSUB8 (intrinsic function) 365–366
 __USADA8 (intrinsic function) 365
 __USAD8 (intrinsic function) 365
 UsageFault_Handler (exception function) 64
 __USAT (intrinsic function) 365
 __USAT16 (intrinsic function) 365
 __USAX (intrinsic function) 364
 --use_c++_inline (compiler option) 258
 --use_unix_directory_separators (compiler option) 258
 __USUB16 (intrinsic function) 364
 __USUB8 (intrinsic function) 364
 utilities (ELF) 417
 utility (STL header file) 378
 __UXTAB (intrinsic function) 366

V

-V (iarchive option) 444
 valaway (library header file) 378
 variables
 auto 54
 defined inside a function 54
 hints for choosing 200
 local. *See* auto variables
 non-initialized 205
 placing at absolute addresses 193
 placing in named sections 193
 static
 placement in memory 53
 taking the address of 201
 variables, global
 placement in memory. 53
 variadic macros 155
 vector floating-point unit 238
 vector (pragma directive) 453, 468
 vector (STL header file) 378
 __vector_table, array holding vector table 64
 veneers 90
 __VER__ (predefined symbol) 371
 --verbose (iarchive option) 444
 --verbose (ielftool option) 444
 version
 of compiler. 371
 of this guide 2
 --vfe (compiler option) 285
 VFP 238
 --vla (compiler option) 258
 void, pointers to 156
 volatile
 and const, declaring objects 300
 declaring objects 299
 protecting simultaneously accesses variables 203
 rules for access. 300

W

#warning message (preprocessor extension)	372
warnings	216
classifying in compiler	233
classifying in linker	269
disabling in compiler	250
disabling in linker	280
exit code in compiler	259
exit code in linker	286
warnings icon, in this guide	27
warnings (pragma directive)	453, 468
--warnings_affect_exit_code (compiler option)	213, 259
--warnings_affect_exit_code (linker option)	286
--warnings_are_errors (compiler option)	259
--warnings_are_errors (linker option)	286
wchar_t (data type), adding support for in C	290
wchar.h (library header file)	376, 379
wctype.h (library header file)	376
__weak (extended keyword)	314
weak (pragma directive)	334
web sites, recommended	25
__WFE (intrinsic function)	366
__WFI (intrinsic function)	366
white-space characters, implementation-defined behavior	445
__write (library function)	117
customizing	113
__write_array, in stdio.h	380
__write_buffered (DLIB library function)	103

X

-x (iarchive option)	435
xreportassert.c	122

Y

__YIELD (intrinsic function)	366
--	-----

Z

zeros, packing algorithm for initializers	392
---	-----

Symbols

__AEABI_PORTABILITY_LEVEL (preprocessor symbol)	184
__AEABI_PORTABLE (preprocessor symbol)	184
_Exit (library function)	110
_exit (library function)	110
__AAPCS_VFP__ (predefined symbol)	368
__AAPCS__ (predefined symbol)	368
__absolute (extended keyword)	307
__ALIGNOF__ (operator)	154
__arm (extended keyword)	307
__ARMVFPV2__ (predefined symbol)	368
__ARMVFPV3__ (predefined symbol)	368
__ARMVFPV4__ (predefined symbol)	368
__ARMVFP_D6__ (predefined symbol)	368
__ARMVFP_FP16__ (predefined symbol)	368
__ARMVFP_SP__ (predefined symbol)	368
__ARMVFP__ (predefined symbol)	368
__ARM_ADVANCED_SIMD__ (predefined symbol)	368
__ARM_MEDIA__ (predefined symbol)	368
__ARM_PROFILE_M__ (predefined symbol)	368
__ARM4M__ (predefined symbol)	369
__ARM4TM__ (predefined symbol)	369
__ARM5E__ (predefined symbol)	369
__ARM5__ (predefined symbol)	369
__ARM6M__ (predefined symbol)	369
__ARM6SM__ (predefined symbol)	369
__ARM6__ (predefined symbol)	369
__ARM7A__ (predefined symbol)	369
__ARM7EM__ (predefined symbol)	369
__ARM7M__ (predefined symbol)	369
__ARM7R__ (predefined symbol)	369
__asm (language extension)	132

__assignment_by_bitwise_copy_allowed, symbol used in library	381	__get_interrupt_state (intrinsic function)	349
__BASE_FILE__ (predefined symbol)	369	__get_IPSR (intrinsic function)	350
__big_endian (extended keyword)	307	__get_LR (intrinsic function)	350
__BUILD_NUMBER__ (predefined symbol)	369	__get_MSP (intrinsic function)	350
__close (library function)	117	__get_PRIMASK (intrinsic function)	351
__CLREX (intrinsic function)	346	__get_PSP (intrinsic function)	351
__CLZ (intrinsic function)	346	__get_PSR (intrinsic function)	351
__code, symbol used in library	381	__get_SB (intrinsic function)	351
__constrange(), symbol used in library	381	__get_SP (intrinsic function)	351
__construction_by_bitwise_copy_allowed, symbol used in library	381	__has_constructor, symbol used in library	381
__CORE__ (predefined symbol)	369	__has_destructor, symbol used in library	381
__cplusplus (predefined symbol)	369	__IAR_DLIB_PERTHREAD_INIT_SIZE (macro)	126
__CPU_MODE__ (predefined symbol)	369	__IAR_DLIB_PERTHREAD_SIZE (macro)	125
__data, symbol used in library	381	__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET (symbolptr)	126
__DATE__ (predefined symbol)	369	__iar_maximum_atexit_calls	86
__disable_fiq (intrinsic function)	347	__iar_program_start (label)	108
__disable_interrupt (intrinsic function)	347	__iar_ReportAssert (library function)	122
__disable_irq (intrinsic function)	347	__IAR_SYSTEMS_ICC__ (predefined symbol)	370
__DLIB_FILE_DESCRIPTOR (configuration symbol)	117	__ICCARM__ (predefined symbol)	370
__DLIB_PERTHREAD (ELF section)	407	__interwork (extended keyword)	308
__DMB (intrinsic function)	347	__intrinsic (extended keyword)	308
__DOUBLE__ (predefined symbol)	369	__irq (extended keyword)	309
__DSB (intrinsic function)	348	__ISB (intrinsic function)	352
__embedded_cplusplus (predefined symbol)	369	__LDC (intrinsic function)	352
__enable_fiq (intrinsic function)	348	__LDCL (intrinsic function)	352
__enable_interrupt (intrinsic function)	348	__LDCL_noidx (intrinsic function)	352
__enable_irq (intrinsic function)	348	__LDC_noidx (intrinsic function)	352
__exit (library function)	110	__LDC2 (intrinsic function)	352
__FILE__ (predefined symbol)	369	__LDC2L (intrinsic function)	352
__fiq (extended keyword)	308	__LDC2L_noidx (intrinsic function)	352
__FUNCTION__ (predefined symbol)	158, 370	__LDC2_noidx (intrinsic function)	352
__func__ (predefined symbol)	158, 370	__LDREX (intrinsic function)	353
__gets, in stdio.h	380	__LDREXB (intrinsic function)	353
__get_BASEPRI (intrinsic function)	348	__LDREXD (intrinsic function)	353
__get_CONTROL (intrinsic function)	349	__LDREXH (intrinsic function)	353
__get_CPSR (intrinsic function)	349	__LINE__ (predefined symbol)	370
__get_FAULTMASK (intrinsic function)	349	__little_endian (extended keyword)	309
__get_FPSCR (intrinsic function)	349	__LITTLE_ENDIAN__ (predefined symbol)	370

__low_level_init	108	__REV (intrinsic function)	357
initialization phase	43	__REVSH (intrinsic function).	357
__low_level_init, customizing	111	__REV16 (intrinsic function)	357
__lseek (library function)	117	__root (extended keyword)	311
__MCR (intrinsic function).	353	__ROPI_ (predefined symbol)	370
__memory_of, symbol used in library	381	__SADD8 (intrinsic function).	357
__MRC (intrinsic function).	354	__SADD16 (intrinsic function).	357
__nested (extended keyword)	309	__SASX (intrinsic function)	357
__noreturn (extended keyword)	310	__sbrel (extended keyword)	306
__no_init (extended keyword)	205, 310	__scanf_args (pragma directive)	331
__no_operation (intrinsic function).	354	__section_begin (extended operator)	155
__open (library function)	117	__section_end (extended operator)	155
__packed (extended keyword).	310	__section_size (extended operator)	155
__pcrel (extended keyword)	306	__SEL (intrinsic function)	357
__PKHBT (intrinsic function).	354	__set_BASEPRI (intrinsic function).	357
__PKHTB (intrinsic function).	355	__set_CONTROL (intrinsic function).	357
__PRETTY_FUNCTION__ (predefined symbol).	370	__set_CPSR (intrinsic function)	358
__printf_args (pragma directive).	329	__set_FAULTMASK (intrinsic function)	358
__program_start (label).	108	__set_FPSCR (intrinsic function)	358
__QADD (intrinsic function)	355	__set_interrupt_state (intrinsic function)	358
__QADD8 (intrinsic function)	355	__set_LR (intrinsic function)	358
__QADD16 (intrinsic function)	355	__set_MSP (intrinsic function)	359
__QASX (intrinsic function).	355	__set_PRIMASK (intrinsic function)	359
__QCFlag (intrinsic function).	355	__set_PSP (intrinsic function)	359
__QDADD (intrinsic function)	355	__set_SB (intrinsic function)	359
__QDOUBLE (intrinsic function).	356	__set_SP (intrinsic function).	359
__QDSUB (intrinsic function)	355	__SEV (intrinsic function)	359
__QFlag (intrinsic function)	356	__SHADD8 (intrinsic function)	360
__QSAX (intrinsic function).	355	__SHADD16 (intrinsic function)	360
__QSUB (intrinsic function).	355	__SHASX (intrinsic function).	360
__QSUB16 (intrinsic function).	355	__SHSAX (intrinsic function).	360
__QSUB8 (intrinsic function).	355	__SHSUB16 (intrinsic function).	360
__ramfunc (extended keyword).	311	__SHSUB8 (intrinsic function).	360
executing in RAM	58	__SMLABB (intrinsic function)	360
__RBIT (intrinsic function)	356	__SMLABT (intrinsic function)	360
__read (library function).	117	__SMLAD (intrinsic function)	360
customizing	113	__SMLADX (intrinsic function).	360
__reset_QC_flag (intrinsic function).	356	__SMLALBB (intrinsic function).	360
__reset_Q_flag (intrinsic function).	356	__SMLALBT (intrinsic function).	360

__SMLALD (intrinsic function)	360	__STC2L (intrinsic function)	362
__SMLALDX (intrinsic function)	360	__STC2L_noidx (intrinsic function)	363
__SMLALTB (intrinsic function)	360	__STC2_noidx (intrinsic function)	363
__SMLALTT (intrinsic function)	360	__STDC_VERSION__ (predefined symbol)	371
__SMLATB (intrinsic function)	360	__STDC__ (predefined symbol)	371
__SMLATT (intrinsic function)	360	__STREX (intrinsic function)	363
__SMLAWB (intrinsic function)	360	__STREXB (intrinsic function)	363
__SMLAWT (intrinsic function)	360	__STREXD (intrinsic function)	363
__SMLSDB (intrinsic function)	360	__STREXH (intrinsic function)	363
__SMLSDB (intrinsic function)	360	__swi (extended keyword)	312
__SMLSDB (intrinsic function)	360	__SWP (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__SWPB (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__SXTAB (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__SXTAB16 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__SXTAH (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__SXTB16 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__task (extended keyword)	313
__SMLSDB (intrinsic function)	360	__thumb (extended keyword)	314
__SMLSDB (intrinsic function)	360	__TIME__ (predefined symbol)	371
__SMLSDB (intrinsic function)	360	__UADD8 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UADD16 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UASX (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UHADD8 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UHADD16 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UHASX (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UHSAX (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UHSUB16 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UHSUB8 (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__UMAAL (intrinsic function)	364
__SMLSDB (intrinsic function)	360	__ungetchar, in stdio.h	380
__SMLSDB (intrinsic function)	360	__UQADD8 (intrinsic function)	365
__SMLSDB (intrinsic function)	360	__UQADD16 (intrinsic function)	365
__SMLSDB (intrinsic function)	360	__UQASX (intrinsic function)	365
__SMLSDB (intrinsic function)	360	__UQSAX (intrinsic function)	365–366
__SMLSDB (intrinsic function)	360	__UQSUB16 (intrinsic function)	365–366
__SMLSDB (intrinsic function)	360	__UQSUB8 (intrinsic function)	365–366
__SMLSDB (intrinsic function)	360	__USADA8 (intrinsic function)	365
__SMLSDB (intrinsic function)	360	__USAD8 (intrinsic function)	365
__SMLSDB (intrinsic function)	360	__USAT (intrinsic function)	365
__SMLSDB (intrinsic function)	360		

__USAT16 (intrinsic function)	365	--all (ielfdump option)	431
__USAX (intrinsic function)	364	--arm (compiler option)	227
__USUB16 (intrinsic function)	364	--basic_heap (linker option)	264
__USUB8 (intrinsic function)	364	--BE32 (linker option)	264
__UXTAB (intrinsic function)	366	--BE8 (linker option)	264
__VA_ARGS__ (preprocessor extension)	151	--bin (ielftool option)	431
__VER__ (predefined symbol)	371	--call_graph (compiler option)	265
__weak (extended keyword)	314	--char_is_signed (compiler option)	227
__WFE (intrinsic function)	366	--char_is_unsigned (compiler option)	228
__WFI (intrinsic function)	366	--checksum (ielftool option)	431
__write (library function)	117	--code (ielfdump option)	433
customizing	113	--config (linker option)	265
__write_array, in stdio.h	380	--config_def (linker option)	265
__write_buffered (DLIB library function)	103	--cpp_init_routine (linker option)	266
__YIELD (intrinsic function)	366	--cpu (compiler option)	228
-D (compiler option)	230	--cpu (linker option)	266
-d (iarchive option)	434	--cpu_mode (compiler option)	229
-e (compiler option)	235	--create (iarchive option)	433
-f (compiler option)	237	--c++ (compiler option)	229
-f (IAR utility option)	435	--c89 (compiler option)	227
-f (linker option)	272	--debug (compiler option)	230
-I (compiler option)	239	--define_symbol (linker option)	267
-l (compiler option)	240	--delete (iarchive option)	434
for creating skeleton code	138	--dependencies (compiler option)	231
-O (compiler option)	250	--dependencies (linker option)	267
-o (compiler option)	251	--diagnostics_tables (compiler option)	233
-o (iarchive option)	437	--diagnostics_tables (linker option)	269
-o (ielfdump option)	437	--diag_error (compiler option)	232
-o (linker option)	281	--diag_error (linker option)	268
-r (compiler option)	230	--diag_remark (compiler option)	232
-r (iarchive option)	439	--diag_remark (linker option)	268
-S (iarchive option)	441	--diag_suppress (compiler option)	232
-s (ielfdump option)	440	--diag_suppress (linker option)	269
-t (iarchive option)	444	--diag_warning (compiler option)	233
-V (iarchive option)	444	--diag_warning (linker option)	269
-x (iarchive option)	435	--discard_unused_publics (compiler option)	233
--aapcs (compiler option)	226	--dlib_config (compiler option)	234
--aeabi (compiler option)	226	--ec++ (compiler option)	235
--align_sp_on_irq (compiler option)	226	--edit (isymexport option)	434

--eec++ (compiler option)	235	--misrac2004 (compiler option)	223
--enable_hardware_workaround (compiler option)	236	--misrac2004 (linker option)	262
--enable_hardware_workaround (linker option)	270	--no_clustering (compiler option)	243
--enable_multibytes (compiler option)	236	--no_code_motion (compiler option)	243
--endian (compiler option)	236	--no_const_align (compiler option)	243
--entry (linker option)	270	--no_cse (compiler option)	244
--enum_is_int (compiler option)	237	--no_dynamic_rtti_elimination (linker option)	277
--error_limit (compiler option)	237	--no_exceptions (compiler option)	244
--error_limit (linker option)	271	--no_exceptions (linker option)	278
--exception_tables (linker option)	271	--no_fragments (compiler option)	245
--export_builtin_config (linker option)	272	--no_fragments (linker option)	278
--extract (iarchive option)	435	--no_inline (compiler option)	245
--extra_init (linker option)	272	--no_library_search (linker option)	278
--fill (ielftool option)	435	--no_locals (linker option)	279
--force_exceptions (linker option)	273	--no_loop_align (compiler option)	245
--force_output (linker option)	273	--no_mem_idioms (compiler option)	246
--fpu (compiler option)	238	--no_path_in_file_macros (compiler option)	246
--guard_calls (compiler option)	239	--no_range_reservations (linker option)	279
--header_context (compiler option)	239	--no_remove (linker option)	279
--ihex (ielftool option)	436	--no_rtti (compiler option)	246
--image_input (linker option)	274	--no_rw_dynamic_init (compiler option)	247
--inline (linker option)	274	--no_scheduling (compiler option)	247
--interwork (compiler option)	239	--no_static_destruction (compiler option)	247
--keep (linker option)	275	--no_strtab (ielfdump option)	436
--legacy (compiler option)	241	--no_system_include (compiler option)	248
--lock_regs (compiler option)	241	--no_typedefs_in_diagnostics (compiler option)	248
--log (linker option)	275	--no_unaligned_access (compiler option)	249
--log_file (linker option)	276	--no_unroll (compiler option)	249
--macro_positions_in_diagnostics (compiler option)	242	--no_veneer (linker option)	280
--mangled_names_in_messages (linker option)	276	--no_vfe (compiler option)	280
--map (linker option)	276	--no_warnings (compiler option)	250
--merge_duplicate_sections (linker option)	277	--no_warnings (linker option)	280
--mfc (compiler option)	242	--no_wrap_diagnostics (compiler option)	250
--migration_preprocessor_extensions (compiler option)	242	--no_wrap_diagnostics (linker option)	281
--misrac (compiler option)	223	--only_stdout (compiler option)	251
--misrac_verbose (compiler option)	223	--only_stdout (linker option)	281
--misrac_verbose (linker option)	262	--option_name (compiler option)	270
--misrac1998 (compiler option)	223	--output (compiler option)	251
--misrac1998 (linker option)	262	--output (iarchive option)	437

--output (ielfdump option)	437	--strip (iobjmanip option)	443
--output (linker option)	281	--strip (linker option)	285
--pi_veneer (linker option)	281	--symbols (iarchive option)	443
--place_holder (linker option)	282	--system_include_dir (compiler option)	257
--predef_macro (compiler option)	251	--thumb (compiler option)	257
--preinclude (compiler option)	252	--toc (iarchive option)	444
--preprocess (compiler option)	252	--use_c++_inline (compiler option)	258
--ram_reserve_ranges (ismexport option)	437	--use_unix_directory_separators (compiler option)	258
--raw (ielfdump] option)	438	--verbose (iarchive option)	444
--redirect (linker option)	282	--verbose (ielftool option)	444
--relaxed_fp (compiler option)	253	--vfe (compiler option)	285
--remarks (compiler option)	254	--vla (compiler option)	258
--remarks (linker option)	283	--warnings_affect_exit_code (compiler option)	213, 259
--remove_section (iobjmanip option)	438	--warnings_affect_exit_code (linker option)	286
--rename_section (iobjmanip option)	439	--warnings_are_errors (compiler option)	259
--rename_symbol (iobjmanip option)	439	--warnings_are_errors (linker option)	286
--replace (iarchive option)	439	.bss (ELF section)	406
--require_prototypes (compiler option)	254	.comment (ELF section)	406
--reserve_ranges (ismexport option)	440	.cstart (ELF section)	407
--ropi (compiler option)	254	.data (ELF section)	407
--rwpi (compiler option)	255	.data_init (ELF section)	407
--search (linker option)	283	.debug (ELF section)	406
--section (compiler option)	255	.exc.text (ELF section)	408
--section (ielfdump option)	440	.iar.debug (ELF section)	406
--self_reloc (ielftool option)	441	.iar.dynexit (ELF section)	408
--semihosting (linker option)	283	.init_array (section)	408
--separate_cluster_for_initialized_variables (compiler option)	256	.intvec (section)	408
--silent (compiler option)	256	.noinit (ELF section)	409
--silent (iarchive option)	441	.preinit_array (section)	409
--silent (ielftool option)	441	.prepreinit_array (section)	409
--silent (linker option)	284	.rel (ELF section)	406
--simple (ielftool option)	442	.rela (ELF section)	406
--skip_dynamic_initialization (linker option)	284	.rodata (ELF section)	410
--srec (ielftool option)	442	.shstrtab (ELF section)	406
--srec-len (ielftool option)	442	.strtab (ELF section)	406
--srec-s3only (ielftool option)	443	.symtab (ELF section)	406
--stack_usage_control (compiler option)	284	.text (ELF section)	410
--strict (compiler option)	256	.textrw (ELF section)	410
--strip (ielftool option)	443	.textrw_init (ELF section)	410

@ (operator)
 placing at absolute address 192
 placing in sections 193
#include files, specifying 211, 239
#warning message (preprocessor extension) 372
%Z replacement string,
implementation-defined behavior 458

Numerics

32-bits (floating-point format) 294
64-bits (floating-point format) 295