IAR C/C++ Compiler
Compiling and Linking

for Advanced RISC Machines Ltd’s
ARM Cores

DARM-9
COPYRIGHT NOTICE

© 1999–2012 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, I-jet, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by 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
Ninth edition: May 2012

Part number: DARM-9

This guide applies to version 6.4x of IAR Embedded Workbench® for ARM®.

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 ............................................................................................ 69
Linking your application ...................................................................................... 85
The DLIB runtime environment .......................................................................... 99
Assembler language interface ............................................................................. 137
Using C ............................................................................................................... 161
Using C++ ......................................................................................................... 171
Application-related considerations .................................................................... 183
Efficient coding for embedded applications ..................................................... 197

Part 2. Reference information .............................................................................. 217
External interface details .................................................................................... 219
Compiler options ............................................................................................... 227
Linker options .................................................................................................... 269
Data representation ............................................................................................. 297
Extended keywords ............................................................................................ 313
Pragma directives ................................................................. 325
Intrinsic functions ............................................................. 345
The preprocessor ............................................................... 381
Library functions .............................................................. 389
The linker configuration file ............................................... 399
Section reference ............................................................. 423
Stack usage control files .................................................... 429
IAR utilities .......................................................................... 437
Implementation-defined behavior for Standard C .............. 467
Implementation-defined behavior for C89 ....................... 483
Index .................................................................................. 495
**Contents**

Tables ...................................................................................................................... 19

Preface .................................................................................................................... 21

- **Who should read this guide** ............................................................... 21
- **How to use this guide** .......................................................................... 21
- **What this guide contains** ................................................................. 21
  Part 1. Using the build tools ......................................................................... 22
  Part 2. Reference information ...................................................................... 22

**Other documentation** ........................................................................... 23

- User and reference guides ........................................................................ 24
- The online help system .............................................................................. 24
- Further reading .......................................................................................... 25
- Web sites .................................................................................................. 25

**Document conventions** ........................................................................ 26

- Typographic conventions ......................................................................... 26
- Naming conventions ................................................................................... 27

**Part 1. 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
   Potential problems ........................................................................ 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
Inlining functions ............................................................................... 65
   C versus C++ semantics ............................................................. 66
   Features controlling function inlining ........................................ 66
Linking using ILINK .......................................................................... 69
   Linking—an overview .............................................................. 69
   Modules and sections ............................................................... 70
   The linking process ................................................................... 71
   Placing code and data—the linker configuration file .................. 73
      A simple example of a configuration file .............................. 73
Initialization at system startup .......................................................... 76
   The initialization process ........................................................... 76
   C++ dynamic initialization ....................................................... 78
Stack usage analysis ......................................................................... 78
   Limitations ............................................................................... 79
   Stack usage control files .......................................................... 79
   Source annotation ..................................................................... 80
   Situations where warnings are issued ...................................... 80
Map file contents ................................................................. 81
Checking that the stack is large enough ............................... 82
Call graph log ................................................................. 82
Call graph XML output ........................................................ 84

**Linking your application** ...................................................... 85

**Linking considerations** ......................................................... 85
Choosing a linker configuration file ........................................ 85
Defining your own memory areas ........................................... 86
Placing sections ................................................................. 87
Reserving space in RAM ........................................................ 88
Keeping modules ............................................................... 89
Keeping symbols and sections ................................................. 89
Application startup .............................................................. 89
Setting up the stack .............................................................. 89
Setting up the heap .............................................................. 90
Setting up the atexit limit ....................................................... 90
Changing the default initialization .......................................... 90
Interaction between ILINK and the application ....................... 94
Standard library handling ...................................................... 94
Producing other output formats than ELF/DWARF ...................... 95
Veneers ........................................................................... 95

**Hints for troubleshooting** ....................................................... 95
Relocation errors ............................................................... 95

The DLIB runtime environment ............................................ 99

**Introduction to the runtime environment** ................................. 99
Runtime environment functionality ........................................ 99
Setting up the runtime environment ........................................ 100

**Using prebuilt libraries** ..................................................... 101
Library filename syntax ......................................................... 102
Groups of library files ........................................................ 103
Customizing a prebuilt library without rebuilding ...................... 104

**Choosing formatters for printf and scanf** .................. 105
Choosing a printf formatter .................................................. 105
Time ........................................................................................................... 127
Math functions ....................................................................................... 127
  Smaller versions ................................................................................. 128
  More accurate versions .................................................................. 129
Assert ......................................................................................................... 129
Atexit ......................................................................................................... 130
Managing a multithreaded environment .................................... 130
  Multithread support in the DLIB library ........................................ 130
  Enabling multithread support ........................................................ 131
  TLS in the linker configuration file ................................................. 135
Checking module consistency ......................................................... 135
  Runtime model attributes ........................................................... 135
  Using runtime model attributes ................................................ 136
Assembler language interface .......................................................... 137
  Mixing C and assembler .................................................................. 137
    Intrinsic functions ....................................................................... 137
    Mixing C and assembler modules .............................................. 138
    Inline assembler ........................................................................... 139
  This is an example of how to use clobbered memory ...................... 147
Calling assembler routines from C .................................................. 147
  Creating skeleton code ................................................................. 147
  Compiling the code ......................................................................... 148
Calling assembler routines from C++ .......................................... 149
Calling convention .............................................................................. 150
  Function declarations ..................................................................... 151
  Using C linkage in C++ source code ............................................ 151
  Preserved versus scratch registers .............................................. 151
  Function entrance .......................................................................... 152
  Function exit ................................................................................... 154
  Examples .......................................................................................... 155
Call frame information .......................................................................... 157
  CFI directives ................................................................................ 157
  Creating assembler source with CFI support ............................... 158
Part 1. Using the build tools

Using C

C language overview ................................................................. 161
Extensions overview ................................................................. 162
IAR C language extensions .......................................................... 163
Extensions for embedded systems programming ......................... 164
Relaxations to Standard C ............................................................ 166

Using C++

Overview—Embedded C++ and Extended
Embedded C++ ............................................................................... 171
Embedded C++ ............................................................................. 171
Extended Embedded C++ .............................................................. 172
Overview—Standard C++ .............................................................. 172
Exception handling ....................................................................... 174
Enabling support for C++ and variants ........................................... 175
C++ and EEC++ feature descriptions .............................................. 176
Using IAR attributes with Classes ................................................. 176
Function types ............................................................................. 176
Using static class objects in interrupts ......................................... 177
Using New handlers ...................................................................... 177
Templates ...................................................................................... 178
Debug support in C-SPY ................................................................. 178
EEC++ feature description ............................................................. 178
Templates ...................................................................................... 179
Variants of cast operators ............................................................. 179
Mutable ......................................................................................... 179
Namespace .................................................................................... 179
The STD namespace ..................................................................... 179
EEC++ and C++ language extensions ............................................. 180
Application-related considerations ........................................... 183
Output format considerations ...................................................... 183
Stack considerations ................................................................. 183
  Stack size considerations ....................................................... 184
  Stack alignment ................................................................. 184
  Exception stack ................................................................. 184
Heap considerations ................................................................. 185
  Heap size and standard I/O .................................................... 185
Interaction between the tools and your application ...................... 186
Checksum calculation ............................................................... 187
  Calculating a checksum ........................................................ 188
  Adding a checksum function to your source code ...................... 189
  Things to remember ............................................................ 191
  C-SPY considerations ......................................................... 192
Linker optimizations ............................................................... 192
  Virtual Function Elimination .................................................. 192
AEABI compliance ................................................................. 192
  Linking AEABI-compliant modules using the IAR ILINK linker ....... 193
  Linking AEABI-compliant modules using a third-party linker ....... 194
  Enabling AEABI compliance in the compiler ............................. 194
CMSIS integration ................................................................. 195
  CMSIS DSP library ............................................................. 195
  Customizing the CMSIS DSP library ....................................... 195
  Building with CMSIS on the command line ............................... 195
  Building with CMSIS in IAR Embedded Workbench .................... 195
Efficient coding for embedded applications ................................ 197
Selecting data types ............................................................... 197
  Using efficient data types ..................................................... 197
  Floating-point types ........................................................... 198
  Alignment of elements in a structure ...................................... 199
  Anonymous structs and unions ............................................. 199
Controlling data and function placement in memory ..................... 201
  Data placement at an absolute location ................................... 202
Part 1. Using the build tools

Contents

13

Data and function placement in sections ........................................ 203
Data placement in registers ......................................................... 204

Controlling compiler optimizations ............................................. 205
Scope for performed optimizations .............................................. 206
Multi-file compilation units ....................................................... 206
Optimization levels ................................................................. 207
Speed versus size .................................................................... 208
Fine-tuning enabled transformations ......................................... 208

Facilitating good code generation ............................................. 211
Writing optimization-friendly source code .................................... 211
Saving stack space and RAM memory ......................................... 211
Function prototypes .............................................................. 212
Integer types and bit negation .................................................... 213
Protecting simultaneously accessed variables .............................. 213
Accessing special function registers .......................................... 214
Passing values between C and assembler objects ...................... 215
Non-initialized variables ......................................................... 215

Part 2. Reference information .................................................. 217

External interface details .......................................................... 219

Invocation syntax ................................................................. 219
Compiler invocation syntax ........................................................ 219
ILINK invocation syntax .......................................................... 219
Passing options ....................................................................... 220
Environment variables ............................................................. 220

Include file search procedure .................................................. 221

Compiler output ....................................................................... 222
Error return codes .................................................................. 223

ILINK output ........................................................................... 223

Diagnostics ............................................................................... 224
Message format for the compiler .............................................. 224
Message format for the linker .................................................. 225
Severity levels ......................................................................... 225
Setting the severity level ......................................................... 226
Internal error ................................................................. 226

Compiler options ........................................................................ 227

Options syntax ........................................................................... 227
Types of options ....................................................................... 227
Rules for specifying parameters ............................................... 227

Summary of compiler options ............................................... 229

Descriptions of compiler options ........................................ 233

Linker options .............................................................................. 269

Summary of linker options ...................................................... 269

Descriptions of linker options ............................................... 271

Data representation ...................................................................... 297

Alignment .................................................................................... 297
  Alignment on the ARM core ..................................................... 297

Byte order ................................................................................... 298

Basic data types .......................................................................... 298
  Integer types .......................................................................... 298
  Floating-point types ............................................................... 304

Pointer types .............................................................................. 305
  Function pointers ..................................................................... 305
  Data pointers .......................................................................... 306
  Casting ..................................................................................... 306

Structure types ............................................................................ 307
  Alignment ................................................................................ 307
  General layout ......................................................................... 307
  Packed structure types .......................................................... 307

Type qualifiers ............................................................................ 309
  Declaring objects volatile ...................................................... 309
  Declaring objects volatile and const ..................................... 310
  Declaring objects const ......................................................... 310

Data types in C++ ......................................................................... 311
Extended keywords ................................................................. 313
  General syntax rules for extended keywords ....................... 313
    Type attributes ................................................................. 313
    Object attributes ............................................................ 315
  Summary of extended keywords .......................................... 315
  Descriptions of extended keywords ..................................... 316
Pragma directives ................................................................. 325
  Summary of pragma directives ............................................. 325
  Descriptions of pragma directives ...................................... 327
Intrinsic functions ............................................................... 345
  Summary of intrinsic functions .......................................... 345
    Intrinsic functions for Neon instructions .......................... 351
  Descriptions of intrinsic functions .................................... 352
The preprocessor ................................................................. 381
  Overview of the preprocessor ............................................ 381
  Description of predefined preprocessor symbols .................. 382
  Descriptions of miscellaneous preprocessor extensions ........ 387
Library functions ............................................................... 389
  Library overview ............................................................. 389
    Header files ...................................................................... 389
    Library object files .......................................................... 390
    Alternative more accurate library functions ....................... 390
    Reentrancy ..................................................................... 390
    The longjmp function ...................................................... 391
  IAR DLIB Library ............................................................... 391
    C header files ............................................................... 391
    C++ header files ............................................................ 392
    Library functions as intrinsic functions ............................. 395
    Added C functionality ..................................................... 395
    Symbols used internally by the library .............................. 397
The linker configuration file ................................................................. 399
  Overview ......................................................................................... 399
  Defining memories and regions ....................................................... 400
  Regions ............................................................................................ 401
  Section handling ............................................................................ 405
  Section selection ............................................................................ 413
  Using symbols, expressions, and numbers ...................................... 417
  Structural configuration ................................................................. 420

Section reference .................................................................................. 423
  Summary of sections ......................................................................... 423
  Descriptions of sections and blocks ................................................ 424

Stack usage control files ..................................................................... 429
  Overview .......................................................................................... 429
  C++ names ....................................................................................... 429
  Stack usage control directives ......................................................... 429
  Syntactic components ...................................................................... 433

IAR utilities ........................................................................................... 437
  The IAR Archive Tool—iarchive ....................................................... 437
    Invocation syntax ............................................................................ 437
    Summary of iarchive commands ................................................... 438
    Summary of iarchive options ......................................................... 439
    Diagnostic messages ...................................................................... 439
  The IAR ELF Tool—ielftool .............................................................. 440
    Invocation syntax ............................................................................ 441
    Summary of ielftool options ........................................................ 441
  The IAR ELF Dumper for arm—ielfdumparm .................................... 442
    Invocation syntax ............................................................................ 442
    Summary of ielfdumparm options .................................................. 443
  The IAR ELF Object Tool—iobjmanip .............................................. 443
    Invocation syntax ............................................................................ 443
    Summary of iobjmanip options ....................................................... 444
    Diagnostic messages ...................................................................... 444
The IAR Absolute Symbol Exporter—\textit{isymexport} \textbf{...... 446}
Invocation syntax \textbf{................................................................. 446}
Summary of isymexport options \textbf{........................................ 447}
Steering files \textbf{.......................................................................... 447}
Diagnostic messages \textbf{................................................................... 449}

\textbf{Descriptions of options} \textbf{........................................... 451}

Implementation-defined behavior for Standard C \textbf{........... 467}

\textbf{Descriptions of implementation-defined behavior} \textbf{........ 467}
J.3.1 Translation \textbf{........................................................................ 467}
J.3.2 Environment \textbf{..................................................................... 468}
J.3.3 Identifiers \textbf{.......................................................................... 469}
J.3.4 Characters \textbf{......................................................................... 469}
J.3.5 Integers \textbf{............................................................................. 470}
J.3.6 Floating point \textbf{..................................................................... 471}
J.3.7 Arrays and pointers \textbf{.......................................................... 472}
J.3.8 Hints \textbf{............................................................................... 472}
J.3.9 Structures, unions, enumerations, and bitfields \textbf{............. 473}
J.3.10 Qualifiers \textbf{......................................................................... 473}
J.3.11 Preprocessing directives \textbf{.................................................... 473}
J.3.12 Library functions \textbf{............................................................. 476}
J.3.13 Architecture \textbf{.................................................................... 480}
J.4 Locale \textbf{................................................................................ 481}

Implementation-defined behavior for C89 \textbf{......................... 483}

\textbf{Descriptions of implementation-defined behavior} \textbf{........ 483}
Translation \textbf{............................................................................. 483}
Environment \textbf{.......................................................................... 483}
Identifiers \textbf{.............................................................................. 484}
Characters \textbf{............................................................................. 484}
Integers \textbf{................................................................................ 485}
Floating point \textbf{......................................................................... 486}
Arrays and pointers \textbf{............................................................... 486}
Registers \textbf{.............................................................................. 487}
Structures, unions, enumerations, and bitfields \textbf{............... 487}
# Tables

1: Typographic conventions used in this guide .......................................................... 26  
2: Naming conventions used in this guide ............................................................... 27  
3: Sections holding initialized data ......................................................................... 76  
4: Description of a relocation error ......................................................................... 96  
5: Customizable items ............................................................................................ 104  
6: Formatters for printf ......................................................................................... 105  
7: Formatters for scanf ......................................................................................... 106  
8: Functions with special meanings when linked with debug library .................... 110  
9: Library configurations ....................................................................................... 118  
10: Descriptions of printf configuration symbols .................................................. 122  
11: Descriptions of scanf configuration symbols ................................................... 122  
12: Low-level I/O files ............................................................................................ 123  
13: Library objects using TLS ................................................................................. 131  
14: Macros for implementing TLS allocation ....................................................... 133  
15: Example of runtime model attributes ................................................................ 135  
16: Inline assembler operand constraints ................................................................ 142  
17: Supported constraint modifiers ......................................................................... 143  
18: Operand modifiers and transformations ............................................................. 144  
19: List of valid clobbers ......................................................................................... 146  
20: Registers used for passing parameters ............................................................... 153  
21: Registers used for returning values ................................................................... 154  
22: Call frame information resources defined in a names block ............................ 157  
23: Language extensions ......................................................................................... 163  
24: Section operators and their symbols ................................................................ 166  
25: Exception stacks ............................................................................................... 184  
26: Compiler optimization levels ............................................................................ 207  
27: Compiler environment variables ....................................................................... 220  
28: ILINK environment variables .......................................................................... 221  
29: Error return codes ............................................................................................ 223  
30: Compiler options summary ............................................................................... 229  
31: Linker options summary ................................................................................... 269
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, 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.
What this guide contains

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.
Part 1. Using the build tools

Preface

- **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 *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*.
- 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:

- Information about debugging using the IAR C-SPY® Debugger
- Information about using the editor
- 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:

- 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.

WEB SITES

Recommended web sites:

- The Advanced RISC Machines Ltd web site, [www.arm.com](http://www.arm.com), that contains information and news about the ARM cores.
- The IAR Systems web site, [www.iar.com](http://www.iar.com), that holds application notes and other product information.
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 6.3\arm\doc.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>• Source code examples and file paths.</td>
</tr>
<tr>
<td></td>
<td>• Text on the command line.</td>
</tr>
<tr>
<td></td>
<td>• Binary, hexadecimal, and octal numbers.</td>
</tr>
<tr>
<td>parameter</td>
<td>A placeholder for an actual value used as a parameter, for example filename.h where filename represents the name of the file.</td>
</tr>
<tr>
<td>[option]</td>
<td>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.</td>
</tr>
<tr>
<td>{option}</td>
<td>A mandatory part of a linker directive, { and } are not part of the actual directive, but any [ ], { }, or } are part of the directive syntax</td>
</tr>
<tr>
<td>[option]</td>
<td>An optional part of a command.</td>
</tr>
<tr>
<td>[a</td>
<td>b</td>
</tr>
<tr>
<td>{a</td>
<td>b</td>
</tr>
<tr>
<td>bold</td>
<td>Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.</td>
</tr>
<tr>
<td>italic</td>
<td>• A cross-reference within this guide or to another guide.</td>
</tr>
<tr>
<td></td>
<td>• Emphasis.</td>
</tr>
</tbody>
</table>

Table 1: Typographic conventions used in this guide
NAMING CONVENTIONS

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

<table>
<thead>
<tr>
<th>Brand name</th>
<th>Generic term</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR Embedded Workbench® for ARM</td>
<td>IAR Embedded Workbench®</td>
</tr>
<tr>
<td>IAR Embedded Workbench® IDE for ARM</td>
<td>the IDE</td>
</tr>
<tr>
<td>IAR C-SPY® Debugger for ARM</td>
<td>C-SPY, the debugger</td>
</tr>
<tr>
<td>IAR C-SPY® Simulator</td>
<td>the simulator</td>
</tr>
<tr>
<td>IAR C/C++ Compiler™ for ARM</td>
<td>the compiler</td>
</tr>
<tr>
<td>IAR Assembler™ for ARM</td>
<td>the assembler</td>
</tr>
<tr>
<td>IAR ILINK Linker™</td>
<td>ILINK, the linker</td>
</tr>
<tr>
<td>IAR DLIB Library™</td>
<td>the DLIB Library</td>
</tr>
</tbody>
</table>

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

Table 2: Naming conventions used in this guide
Part 1. 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 IAR Assembler Reference Guide for ARM.

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—isymexport—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:

- 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 IAR Assembler Reference Guide for ARM.

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 242 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 chapterPragma 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 137.
Special support for embedded systems
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 201. 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 73.
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 214.

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](http://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 192.

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 192.

---

**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 IAR Assembler Reference Guide for ARM.

This illustration shows the translation process:

![Translation Process Diagram](image)

**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](image)

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.
Application execution—an overview

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

- Initialization phase

For more information about the procedure performed by ILINK, see The linking process, page 71.

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 440.

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

![Diagram showing possible uses of the absolute output ELF/DWARF file](image)

Application execution—an overview

This section gives an overview of the execution of an embedded application divided into three phases, the:
Part 1. Using the build tools

Developing embedded applications

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 `startup.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.

- 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](image)
Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

```
int i = 0;
```

**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:

```
vector
  • Jump to csrtup

csrtup
  • Set up stack pointer
  • Initialize variables to zero
  • Initialize variables
  • Call main()

main( ) and other code

Initializers

Stack

Zero-initialized variables

Initialized variables
```

Figure 6: Initializing variables

4 Finally, the `main` function is called:

```
vector
  • Jump to csrtup

csrtup
  • Set up stack pointer
  • Initialize variables to zero
  • Initialize variables
  • Call main()

main( ) and other code

Initializers

Stack

Zero-initialized variables

Initialized variables
```

Figure 7: Calling main
For more information about each stage, see *System startup and termination*, page 113.
For more information about initialization of data, see *Initialization at system startup*, page 76.

**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 116.

---

**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 --arm, page 234 and --thumb, page 265, 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 also Veneers, page 95.
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 248, 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 specify the ARM core; see **--fpu**, page 246 for syntax information.

**Byte order**
The compiler 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 244, for syntax information.

**OPTIMIZATION FOR SPEED AND SIZE**
The compiler’s optimizer performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression 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.

Two different sets of runtime libraries are provided:

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 for both libraries, 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 environments, see the chapters The DLIB runtime environment, respectively.

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 118, 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 application-specific linker options or the include path to application-specific header files by using the -I option, for example:

-I MyApplication\inc
For information about the prebuilt library object files, see *Using prebuilt libraries*, page 101 (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 105 (DLIB).
- The size of the stack and the heap, see *Setting up the stack*, page 89, and *Setting up the heap*, page 90, 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 either in registers or 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. For more information, see *Auto variables—on the stack*, page 54.
  - 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.

\[
\text{int } \ast \text{MyFunction}() \\
\{ \\
\text{int } x; \\
\text{/* Do something here. */} \\
\text{return } \&x; \text{/* Incorrect */} \\
\}
\]

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 \texttt{malloc}, or one of the related functions \texttt{calloc} and \texttt{realloc}. The memory is released again using \texttt{free}.

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

For related information, see \textit{--basic_heap}, page 272.

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
- Inline functions
- 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 197. 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 0x0-0xFFFFFPPP 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 305.
Execution in RAM

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 113.

The keyword is specified before the return type:

```c
__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:

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

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

can be rewritten to:

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

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

    msg(hello);
}
```

For more information, see *Initializing code—copying ROM to RAM*, page 93.
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 information, 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 \texttt{__irq} or the \texttt{__fiq} keyword can be used. For example:

\begin{verbatim}
__irq __arm void IRQ_Handler(void)
{
    /* Do something */
}
\end{verbatim}

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

**Note:** An interrupt function must have the return type \texttt{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 \texttt{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 \texttt{cstartup.s} and referred to by the library exception vector code:

- \texttt{Undefined_Handler}
- \texttt{SWI_Handler}
- \texttt{Prefetch_Handler}
- \texttt{Abort_Handler}
- \texttt{IRQ_Handler}
- \texttt{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 \texttt{IRQ_Handler}:

\begin{verbatim}
__irq __arm void IRQ_Handler()
{
}
\end{verbatim}
An interrupt function must have C linkage, read more in Calling convention, page 150.

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

```c
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 318 and `type_attribute`, page 342, 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 318, and `type_attribute`, page 342, 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):

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

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

```c
#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 150.

For more information, see __swi, page 322, and swi_number, page 341, 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 322.

**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 89.

**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 215.

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();
};

Inlining functions

Function inlining means that a function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the function call. This optimization, which is performed at optimization level High, normally reduces execution time, but might increase the code size. The resulting code might become more difficult to debug. Whether the inlining actually occurs is subject to the compiler’s heuristics.

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.
**C VERSUS C++ SEMANTICS**

In C++, all definitions of a specific inline function in separate translation units must be exactly the same. If the function is not inlined in one or more of the translation units, then one of the definitions from these translation units will be used as the function implementation.

In C, you must manually select one translation unit that includes the non-inlined version of an inline function. You do this by explicitly declaring the function as `extern` in that translation unit. If you declare the function as `extern` in more than one translation unit, the linker will issue a *multiple definition* error. In addition, in C, inline functions cannot refer to static variables or functions.

For example:

```c
// In a header file.
static int sX;
inline void F(void)
{
    //static int sY; // Cannot refer to statics.
    //sX;            // Cannot refer to statics.
}

// In one source file.
// Declare this F as the non-inlined version to use.
extern inline void F();
```

**FEATURES CONTROLLING FUNCTION INLINING**

There are several mechanisms for controlling function inlining:

- **The `inline` keyword** advises the compiler that the function defined immediately after the directive should be inlined.
  
  If you compile your function in C or C++ mode, the keyword will be interpreted according to its definition in Standard C or Standard C++, respectively.
  
  The main difference in semantics is that in Standard C you cannot (in general) simply supply an inline definition in a header file. You must supply an external definition in one of the compilation units, by designating the inline definition as being external in that compilation unit.

- **`#pragma inline`** is similar to the `inline` keyword, but with the difference that the compiler always uses C++ inline semantics.
  
  By using the `#pragma inline` directive you can also disable the compiler’s heuristics to either force inlining or completely disable inlining. For more information, see `inline`, page 332.
• `--use_c++_inline` forces the compiler to use C++ semantics when compiling a Standard C source code file.

• `--no_inline`, `#pragma optimize=no_inline`, and `#pragma inline=never` all disable function inlining. By default, function inlining is enabled at optimization level High.

The compiler can only inline a function if the definition is known. Normally, this is restricted to the current translation unit. However, when the `--mfc` compiler option for multi-file compilation is used, the compiler can inline definitions from all translation units in the multi-file compilation unit. For more information, see Multi-file compilation units, page 206.

For more information about the function inlining optimization, see Function inlining, page 209.
Inlining functions
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 95.

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 DW ARF 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 89, and Keeping symbols and sections, page 89.

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](image)

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 `ielfdumparm`. See *The IAR ELF Dumper for arm—ielfdumparm*, page 442.
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 */
Placing code and data—the linker configuration file

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

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

The file then creates an empty block called $\text{STACK}$ 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 $\text{.cstartup}$—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 $\{ \text{readonly section .cstartup } \}$ takes precedence over the more generic section selection $\{ \text{readonly } \}$.

Finally, the read/write (readwrite) sections and the $\text{STACK}$ block are placed in the RAM region.
This illustration gives a schematic overview of how the application is placed in memory:

![Application in memory diagram]

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:

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

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:

```c
/* 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 408), 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 85.
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:

```c
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 413.

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.
You can use the `check` directive in your linker configuration file to check that the stack usage calculated by the linker does not exceed the stack space you have allocated.

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. In particular, this might be a problem with object modules written in assembler or produced by non-IAR tools.
- 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 that uses exceptions is not supported.
- If you use other forms of function calls, like software interrupts, they will not be reflected in the call graph.
- Using multi-file compilation (`--mfc`) can interfere with using a stack usage control file to specify properties of module-local functions in the involved files.

Note that stack usage analysis produces a worst case result. The program might not actually ever end up in the maximum call chain, by design, or by coincidence. In particular, the set of possible destinations for a virtual function call in C++ might sometimes include implementations of the function in question which cannot, in fact, be called from that point in the code.

Stack usage analysis is only a complement to actual measurement. If the result is important, you need to perform independent validation of the results of the 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`.

Specify a maximum recursion depth for a recursion nest (a set of cycles in the call graph with at least one common node).

Selectively suppress the warning about unmentioned functions referenced by a module for which you have supplied stack usage information in the stack usage control file.

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:

```c
call_graph_root [interrupt]: Irq1Handler, Irq2Handler;
```

For more information, see `call_graph_root`, page 328 and the chapter `Stack usage control files`, page 429.

**SOURCE ANNOTATION**

As an alternative to specifying possible calls in a stack usage control file, you can instead annotate the source code.

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.

**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) for which no maximum recursion depth has been supplied, or which is of a form for which the linker is unable to calculate a reliable estimate of stack usage.
- There are calls to a function declared as a call graph root.
- You have used the stack usage control file to supply stack usage information for functions in a module that does not have such information, and there are functions
referenced by that module which have not been mentioned as being called in the stack usage control file.

MAP FILE CONTENTS

When stack usage analysis is enabled, the linker map file contains a stack usage chapter with a summary of the stack usage for each call graph root category, and 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
***

Call Graph Root Category  Max Use  Total Use
-------------------------------------------  --------  ---------
interrupt                  104       136
Program entry              168       168

Program entry
"__ iar_program_start": 0x000085ac
Maximum call chain        168 bytes

"__ iar_program_start"
"__cmain"
"main"                   8
"printf"                  24
"_PrintfTiny"             56
"_Prout"                  16
"putchar"                 16
"_write"                  0
"_dwrite"                 0
"__ iar_sh_stdout"        24
"__ iar_get_ttio"         24
"__ iar_lookup_ttioh"     0

interrupt
"FaultHandler": 0x00008434

Maximum call chain       32 bytes

"FaultHandler"            32
interrupt
  "IRQHandler": 0x00008424

Maximum call chain                     104 bytes
  "IRQHandler"                          24
  "do_something" in suexample.o [1]     80

The summary contains the depth of the deepest call chain in each category as well as the
sum of the depths of the deepest call chains in that category.

In this case, the maximum stack depth for the program entry (__iar_program_start) is
168 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
do_something above).

CHECKING THAT THE STACK IS LARGE ENOUGH
You can use the check that directive in your linker configuration file to check that the
stack is large enough.

For example:

      check that size(block CSTACK) >= maxstack("Program entry")
           + totalstack("interrupt") + 100;

When linking, the linker emits an error if the expression is false (zero). In this example
there would be an error if the sum of 168 (the maximum stack usage of the program
entry), 136 (the sum of the maximum stack usages in category "interrupt"), and 100 (a
safety margin) is greater than the size of the CSTACK block.

CALL GRAPH LOG
To help you interpret the results of the stack usage analysis, there is a log output option
that produces a simple text representation of the call graph (--log call_graph).
Example output:

```
Program entry:
0 __iar_program_start [168]
  0 __cmain [168]
    0 __iar_data_init3 [16]
      8 __iar_zero_init3 [8]
        16 - [0]
      8 __iar_copy_init3 [8]
        16 - [0]
    0 __low_level_init [0]
  0 main [168]
    8 printf [160]
      32 __PrintfTiny [136]
      88 _Prout [80]
      104 putchar [64]
      120 _write [48]
      120 __dwrite [48]
      120 __iar_sh_stdout [48]
        144 __iar_get_ttio [24]
          168 __iar_lookup_ttioh [0]
      120 __iar_sh_write [24]
        144 - [0]
      88 __aeabi_uidiv [0]
      88 __aeabi_idiv0 [0]
      88 strlen [0]
    0 exit [8]
      0 _exit [8]
      0 __exit [8]
    0 __iar_close_ttio [8]
      8 __iar_lookup_ttioh [0] ***
    0 __exit [8] ***
```

Each line consists of this information:

- The stack usage at the point of call of the function
- The name of the function, or a single '-' to indicate usage in a function at a point with no function call (typically in a leaf function)
- The stack usage along the deepest call chain from that point. If no such value could be calculated, "[---]" is output instead. "***" marks functions that have already been shown.
CALL GRAPH XML 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 files:

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

The 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.

For some devices, device-specific configuration files are automatically selected.
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.

Do not 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:

```c
/* 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:

```c
/* 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:

```c
/* 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

```c
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:

```c
/* 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:

```c
/* Place section .vectors at address 0 */
place at address Mem:0x0 {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:

```c
/* 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:

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

This is the corresponding example in assembler language:

```asm
    name    createSection
    section MYOWNSECTION:CONST  ; Create a section,
                                   ; and fill it with
        dcl16     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:

```c
/* 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:

```c
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:

```c
    /* Define 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 437.

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 70.

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 71.

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 279.

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 183.

**SETTING UP THE HEAP**

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

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

Specify the appropriate size for your application. If you use a heap, you must allocate at least 50 bytes for it.

**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:

```c
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:

- Suppressing initialization
- 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`).

**Suppressing initialization**

If you do not want the linker to arrange for initialization by copying, for some or all sections, make sure that those sections do not match a pattern in an `initialize by copy` directive (or use an `except` clause to exclude them from matching). If you do not want any initialization by copying at all, you can omit the `initialize by copy` directive entirely.

This can be useful if your application, or just your variables, are loaded into RAM by some other mechanism before application startup.
Choosing a 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 408.

Manual initialization

In the usual case, the initialize by copy directive is used for making the linker arrange for initialization by copying (with or without packing) of sections with content at application startup. The linker achieves this by logically creating an initialization section for each such section, holding the content of the section, and turning the original section into a section without content. Then, the linker adds table elements to the initialization table so that the initialization will be performed at application startup. You can use initialize manually to suppress the creation of table elements to take control over when and how the elements are copied. This is useful for overlays, but also in a number of other circumstances.

For sections without content (zero-initialized sections), the situation is reversed. The linker arranges for zero initialization of all such sections at application startup, except for those that have been mentioned in a do not initialize directive. Usually, only .noinit sections are specified in a do not initialize directive, but you can add any zero-initialized sections you like, and take direct control over when and how these sections are initialized.

Simple copying example with an implicit block

Assume that you have some initialized variables in MYSECTION. If you add this directive to your linker configuration file:

```
initialize manually { section MYSECTION };
```

you can use this source code example to initialize the section:

```
#pragma section = "MYSECTION"
#pragma section = "MYSECTION_init"

void DoInit()
{
    char * from = __section_begin("MYSECTION_init");
    char * to = __section_begin("MYSECTION");
    memcpy(to, from, __section_size("MYSECTION"));
}
```
This piece of source code takes advantage of the fact that if you use __section_begin (and related operators) with a section name, a synthetic block is created by the linker for those sections.

**Example with explicit blocks**

Assume that you instead of needing manual initialization for variables in a specific section, you need it for all initialized variables from a particular library. In that case, you must create explicit blocks for both the variables and the content. Like this:

```c
initialize manually { section .data object mylib.a };  
define block MYBLOCK { section .data object mylib.a };  
define block MYBLOCK_init { section .data_init object mylib.a };  
```

You must also place the two new blocks using one of the section placement directives, the block MYBLOCK in RAM and the block MYBLOCK_init in ROM.

Then you can initialize the sections using the same source code as in the previous example, only with MYBLOCK instead of MYSECTION.

**Overlay example**

This is a simple overlay example that takes advantage of automatic block creation:

```c
initialize manually { section MYOVERLAY* };  
define overlay MYOVERLAY { section MYOVERLAY1 };  
define overlay MYOVERLAY { section MYOVERLAY2 };  
```

You must also place overlay MYOVERLAY somewhere in RAM. The copying could look like this:

```c
#pragma section = "MYOVERLAY"  
#pragma section = "MYOVERLAY1_init"  
#pragma section = "MYOVERLAY2_init"  

void SwitchToOverlay1()
{
    char * from = __section_begin("MYOVERLAY1_init");
    char * to = __section_begin("MYOVERLAY");
    memcpy(to, from, __section_size("MYOVERLAY1_init"));
}

void SwitchToOverlay2()
{
    char * from = __section_begin("MYOVERLAY2_init");
    char * to = __section_begin("MYOVERLAY");
    memcpy(to, from, __section_size("MYOVERLAY2_init"));
}
```
Initializing code—copying ROM to RAM

Sometimes, an application copies pieces of code from flash/ROM to RAM. You can direct the linker to arrange for this to be done automatically at application startup, or do it yourself at some later time using the techniques described in Manual initialization, page 91.

You need to list the code sections that should be copied in an initialize by copy directive. The easiest way is usually to place the relevant functions in a particular section (for example, RAMCODE), and add section RAMCODE to your initialize by copy directive. For example:

```
initialize by copy { rw, section RAMCODE };
```

If you need to place the RAMCODE functions in some particular location, you must mention them in a placement directive, otherwise they will be placed together with other read/write sections.

If you need to control the manner and/or time of copying, you must use an initialize manually directive instead. See Manual initialization, page 91.

If the functions need to run without accessing the flash/ROM, you can use the __ramfunc keyword when compiling. See Execution in RAM, page 58.

Running all code from RAM

If you want to copy the entire application from ROM to RAM at program startup, use the initialize 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 408.

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:

```c
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 186.

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 186.

### 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 101.
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 440.

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. The called function must be an interworking function, see *Interworking*, page 48. 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 289.

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 284

● The *--map* option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see *--map*, page 285.

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:

<table>
<thead>
<tr>
<th>Message entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kind</strong></td>
<td>The relocation directive that failed. The directive depends on the instruction used.</td>
</tr>
<tr>
<td><strong>Location</strong></td>
<td>The location where the problem occurred, described with the following details:</td>
</tr>
<tr>
<td></td>
<td>• The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, 0x40000448 and &quot;myfunc&quot; + 0x2c.</td>
</tr>
<tr>
<td></td>
<td>• The module, and the file. In this example, the module somecode.o.</td>
</tr>
<tr>
<td></td>
<td>• The section number and section name. In this example, section number 7 with the name .text.</td>
</tr>
<tr>
<td></td>
<td>• The offset, specified in number of bytes, in the section. In this example, 0x2c.</td>
</tr>
<tr>
<td><strong>Destination</strong></td>
<td>The target of the instruction, described with the following details:</td>
</tr>
<tr>
<td></td>
<td>• The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, 0x9000000c and &quot;read&quot; (thus, no offset).</td>
</tr>
<tr>
<td></td>
<td>• The module, and when applicable the library. In this example, the module read.o and the library iolib.a.</td>
</tr>
<tr>
<td></td>
<td>• The section number and section name. In this example, section number 6 with the name .text.</td>
</tr>
<tr>
<td></td>
<td>• The offset, specified in number of bytes, in the section. In this example, 0x0.</td>
</tr>
</tbody>
</table>

*Table 4: Description of a relocation error*
Possible solutions

In this case, the distance from the instruction in \texttt{myfunc} to \texttt{__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 395.
- Special compiler support, for instance functions for switch handling or integer arithmetics.
For more information about AEABI compliance, see AEABI compliance, page 192.

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 101.

- 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 118.

- 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 105. You can also specify the size and placement of the stacks and the heap, see Setting up the stack, page 89, and Setting up the heap, page 90, 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 107.

- 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 110.
Part 1. Using the build tools

The DLIB runtime environment

● 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 111.

● 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 113 and Customizing system initialization, page 117.

● 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 112.

● 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 130.

● Check module consistency
You can use runtime model attributes to ensure that modules are built using compatible settings, see Checking module consistency, page 136.

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 from these elements:

{architecture} is the name of the architecture. It can be one of 4t, 5E, 6M, or 7M for the ARM architectures v4T, v5TE, v6M, or v7M, respectively. Libraries built for the v5TE architecture are also used for the v6 architecture and later (except for v6M and v7M).

{cpu_mode} is one of t or a, for Thumb and ARM, respectively.

{byte_order} is one of l or b for little-endian and big-endian, respectively.

{fp_implementation} is _ when the library is compiled without VFP support, that is, using a software implementation floating-point functions, and v 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 base standard. The linker will use the VFP calling convention, and the base standard entry for other modules.

{language} is c when the library is compiled for Standard C++ support, and e when compiled for Embedded C++ support.

{lib_config} is one of n or f for normal and full, respectively.

(debug_interface) is one of s, b or i, for the SWI/SVC mechanism, the BKPT mechanism, and the IAR-specific breakpoint mechanism, respectively. For more information, see --semihosting, page 292.

(rwpi) is s when the library contains read/write position-independent code, see --rwpi, page 263.

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}(a).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 that are 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:

<table>
<thead>
<tr>
<th>Items that can be customized</th>
<th>Described in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatters for printf and scanf</td>
<td>Choosing formatters for printf and scanf, page 105</td>
</tr>
<tr>
<td>Startup and termination code</td>
<td>System startup and termination, page 113</td>
</tr>
<tr>
<td>Low-level input and output</td>
<td>Standard streams for input and output, page 119</td>
</tr>
<tr>
<td>File input and output</td>
<td>File input and output, page 123</td>
</tr>
<tr>
<td>Low-level environment functions</td>
<td>Environment interaction, page 126</td>
</tr>
<tr>
<td>Low-level signal functions</td>
<td>Signal and raise, page 127</td>
</tr>
<tr>
<td>Low-level time functions</td>
<td>Time, page 127</td>
</tr>
<tr>
<td>Some library math functions</td>
<td>Math functions, page 127</td>
</tr>
<tr>
<td>Size of heaps, stacks, and sections</td>
<td>Stack considerations, page 183</td>
</tr>
<tr>
<td></td>
<td>Heap considerations, page 185</td>
</tr>
<tr>
<td></td>
<td>Placing code and data—the linker configuration file, page 73</td>
</tr>
</tbody>
</table>

*Table 5: Customizable items*

For information about how to override library modules, see *Overriding library modules*, page 111.
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 121.

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:

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

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 121.
Choosing formatters for printf and scanf

**Manually specifying the print formatter in the IDE**
To specify a formatter manually, choose Project>Options and select the General Options category. Select the appropriate option on the Library options page.

**Manually specifying the printf formatter from the command line**
To specify a formatter manually, 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:

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

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 121.
Manually specifying the scanf formatter in the IDE
To specify a formatter manually, choose Project>Options and select the General Options category. Select the appropriate option on the Library options page.

Manually specifying the scanf formatter from the command line
To specify a formatter manually, 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 (--semihosted) 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
```
Adapting the library for target hardware

LOW-LEVEL FUNCTIONS IN THE DEBUG LIBRARY

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

<table>
<thead>
<tr>
<th>Function in DLIB low-level interface</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>Exits the application</td>
</tr>
<tr>
<td>clock</td>
<td>Returns the clock on the host computer</td>
</tr>
<tr>
<td>__close</td>
<td>Closes the associated host file on the host computer</td>
</tr>
<tr>
<td>__exit</td>
<td>Notifies that the end of the application was reached</td>
</tr>
<tr>
<td>__iar_ReportAssert</td>
<td>Prints an assert message to terminal I/O</td>
</tr>
<tr>
<td>__lseek</td>
<td>Searches in the associated host file on the host computer</td>
</tr>
<tr>
<td>__open</td>
<td>Opens a file on the host computer</td>
</tr>
<tr>
<td>__read</td>
<td>Directs stdin to the Terminal I/O window; all other files will read the associated host file</td>
</tr>
<tr>
<td>remove</td>
<td>Removes a file on the host computer</td>
</tr>
<tr>
<td>rename</td>
<td>Renames a file on the host computer</td>
</tr>
<tr>
<td>system</td>
<td>Writes a message to the Debug Log window and returns -1</td>
</tr>
<tr>
<td>time</td>
<td>Returns the time on the host computer</td>
</tr>
<tr>
<td>__write</td>
<td>Directs stdout and stderr to the Terminal I/O window; all other files will write to the associated host file</td>
</tr>
</tbody>
</table>

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

You should not use the low-level interface functions prefixed with _ or __ directly in your application. Instead you should use the high-level functions that use these functions. For more information, see Library low-level interface, page 111.

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 111.
LIBRARY LOW-LEVEL INTERFACE

The library uses a set of low-level functions to communicate with the target system. For example, \texttt{printf} and all other standard output functions use the low-level function \texttt{__write} to send the actual characters to an output device. Most of the low-level functions, like \texttt{__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 \textit{The debug library functionality}, page 108.

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 \texttt{stdout}, you should use standard library functions like \texttt{printf} or \texttt{puts}, instead of \texttt{__write}.

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

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

- \textit{Standard streams for input and output}, page 119
- \textit{File input and output}, page 123
- \textit{Signal and raise}, page 127
- \textit{Time}, page 127
- \textit{Assert}, page 130.

Overriding library modules

To use a library low-level interface that you have implemented, add it to your application. See \textit{Adapting the library for target hardware}, page 110. 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.
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.

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.

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 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 117.

**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:

- **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:

- **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 76.
- 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:

- 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.

**Figure 10: System termination phase**

An application can terminate normally in two different ways:

- Return from the `main` function
- Call the `exit` function.
If you want your application to do anything extra at exit, for example resetting the system, you can write your own implementation of the \texttt{__exit(int)} function.

\section*{C-SPY interface to system termination}

If your project is linked with the semihosted interface, the normal \texttt{__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 \textit{Application debug support}, page 107.

\section*{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 \texttt{cstartup}.

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

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

\textbf{Note}: Normally, you do not need to customize either of the files \texttt{cmain.s} or \texttt{cexit.s}.

If you intend to rebuild the library, the source files are available in the template library project, see \textit{Building and using a customized library}, page 112.

\textbf{Note}: Regardless of whether you modify the routine \texttt{__low_level_init} or the file \texttt{cstartup.s}, you do not have to rebuild the library.

\section*{\texttt{__LOW_LEVEL_INIT}}

Two skeleton low-level initialization files are supplied with the product: a C source file, \texttt{low_level_init.c} and an alternative assembler source file, \texttt{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 \texttt{__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 111.

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 279.

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:

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

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 242.
- 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 112.

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 110.

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 112. 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 107.
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:

```c
#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:

```c
#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 201.

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 105.

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**

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

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**

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

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 112.
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 110.

Note that file I/O capability in the library is only supported by libraries with the full library configuration, see Library configurations, page 118. 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:

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

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 107.

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:

```c
#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 112.

**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 `LCCATEGORY`. 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`
- `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:

```c
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 111.

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 112.
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 107.

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 111.

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 112.

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 396.

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 111.

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 112.

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 107.

Math functions

Some library math functions are also available size-optimized versions, and in more accurate versions.
SMALLER VERSIONS

The functions \texttt{cos}, \texttt{exp}, \texttt{log}, \texttt{log10}, \texttt{pow}, \texttt{sin}, \texttt{tan}, and \texttt{__iar_Sin} (a help function for \texttt{sin} and \texttt{cos}) exist in additional, smaller versions in the library. They are about 20\% smaller and about 20\% faster than the default versions. The functions handle INF and NaN values. The drawbacks are that they almost always lose some precision and they do not have the same input range as the default versions.

The names of the functions are constructed like:

\[\texttt{__iar}_{\texttt{xxx}}\texttt{\_small<f|l>}\]

where \texttt{f} is used for float variants, \texttt{l} is used for long double variants, and no suffix is used for double variants.

To use these functions, the default function names must be redirected to these names when linking, using the following options:

- \texttt{--redirect sin=\_iar\_sin\_small}
- \texttt{--redirect cos=\_iar\_cos\_small}
- \texttt{--redirect tan=\_iar\_tan\_small}
- \texttt{--redirect log=\_iar\_log\_small}
- \texttt{--redirect log10=\_iar\_log10\_small}
- \texttt{--redirect exp=\_iar\_exp\_small}
- \texttt{--redirect pow=\_iar\_pow\_small}
- \texttt{--redirect \_iar\_Sin=\_iar\_Sin\_small}

- \texttt{--redirect sin=\_iar\_sin\_smallf}
- \texttt{--redirect cos=\_iar\_cos\_smallf}
- \texttt{--redirect tan=\_iar\_tan\_smallf}
- \texttt{--redirect log=\_iar\_log\_smallf}
- \texttt{--redirect log10=\_iar\_log10\_smallf}
- \texttt{--redirect exp=\_iar\_exp\_smallf}
- \texttt{--redirect pow=\_iar\_pow\_smallf}
- \texttt{--redirect \_iar\_Sin=\_iar\_Sin\_smallf}

- \texttt{--redirect sin=\_iar\_sin\_smalll}
- \texttt{--redirect cos=\_iar\_cos\_smalll}
- \texttt{--redirect tan=\_iar\_tan\_smalll}
- \texttt{--redirect log=\_iar\_log\_smalll}
- \texttt{--redirect log10=\_iar\_log10\_smalll}
- \texttt{--redirect exp=\_iar\_exp\_smalll}
- \texttt{--redirect pow=\_iar\_pow\_smalll}
- \texttt{--redirect \_iar\_Sin=\_iar\_Sin\_smalll}

Note that if \texttt{cos} or \texttt{sin} is redirected, \texttt{\_iar\_Sin} must be redirected as well.
MORE ACCURATE VERSIONS

The functions \texttt{cos}, \texttt{pow}, \texttt{sin}, and \texttt{tan}, and the help functions \texttt{__iar\_Sin} and \texttt{__iar\_Pow} exist in versions in the library that are more exact and can handle larger argument ranges. The drawback is that they are larger and slower than the default versions.

The names of the functions are constructed like:

\texttt{__iar\_xxx\_accurate<f|l>}

where \texttt{f} is used for \texttt{float} variants, \texttt{l} is used for \texttt{long double} variants, and no suffix is used for \texttt{double} variants.

To use these functions, the default function names must be redirected to these names when linking, using the following options:

- \texttt{--redirect sin=\_iar\_sin\_accurate}
- \texttt{--redirect cos=\_iar\_cos\_accurate}
- \texttt{--redirect tan=\_iar\_tan\_accurate}
- \texttt{--redirect pow=\_iar\_pow\_accurate}
- \texttt{--redirect __iar\_Sin=\_iar\_Sin\_accurate}
- \texttt{--redirect __iar\_Pow=\_iar\_Pow\_accurate}
- \texttt{--redirect sinf=\_iar\_sin\_accuratef}
- \texttt{--redirect cosf=\_iar\_cos\_accuratef}
- \texttt{--redirect tanf=\_iar\_tan\_accuratef}
- \texttt{--redirect powf=\_iar\_pow\_accuratef}
- \texttt{--redirect __iar\_Sinf=\_iar\_Sin\_accuratef}
- \texttt{--redirect __iar\_Powf=\_iar\_Pow\_accuratef}
- \texttt{--redirect sinl=\_iar\_sin\_accuratel}
- \texttt{--redirect cosl=\_iar\_cos\_accuratel}
- \texttt{--redirect tanl=\_iar\_tan\_accuratel}
- \texttt{--redirect powl=\_iar\_pow\_accuratel}
- \texttt{--redirect __iar\_Sinl=\_iar\_Sin\_accuratel}
- \texttt{--redirect __iar\_Powl=\_iar\_Pow\_accuratel}

Note that if \texttt{cos} or \texttt{sin} is redirected, \texttt{__iar\_Sin} must be redirected as well. The same applies to \texttt{pow} and \texttt{__iar\_Pow}.

**Assert**

If you linked your application with support for runtime debugging, an assert will print a message on \texttt{stdout}. If this is not the behavior you require, you must add the source file \texttt{xreportassert.c} to your application project. The \texttt{__iar\_ReportAssert} function generates the assert notification. You can find template code in the \texttt{arm\src\lib}.
directory. For more information, see *Building and using a customized library*, page 112.

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 388.

---

**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 90.

---

**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.
Part 1. Using the build tools

The DLIB runtime environment

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:

<table>
<thead>
<tr>
<th>Library objects using TLS</th>
<th>When these functions are used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error functions</td>
<td><code>errno</code>, <code>strerror</code></td>
</tr>
<tr>
<td>Locale functions</td>
<td><code>localeconv</code>, <code>setlocale</code></td>
</tr>
<tr>
<td>Time functions</td>
<td><code>asctime</code>, <code>localtime</code>, <code>gmtime</code>, <code>mktime</code></td>
</tr>
<tr>
<td>Multibyte functions</td>
<td><code>mbrolen, mbtowc, mbstowc, mbtowc, wcrtomb, wcsrtomb, wctomb</code></td>
</tr>
<tr>
<td>Rand functions</td>
<td><code>rand, srand</code></td>
</tr>
<tr>
<td>Miscellaneous functions</td>
<td><code>atexit, strtok</code></td>
</tr>
<tr>
<td>C++ exception engine</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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`. 
Managing a multithreaded environment

System locks interface

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

```c
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:

```c
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 DLIB runtime environment

Part 1. Using the build tools

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
- 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:

```c
void *__iar_dlib_perthread_access(void *symbp);
```

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.

```c
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:

```c
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:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__IAR_DLIB_PERTHREAD_SIZE</td>
<td>The size needed for the TLS memory area.</td>
</tr>
<tr>
<td>__IAR_DLIB_PERTHREAD_INIT_SIZE</td>
<td>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.</td>
</tr>
<tr>
<td>__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(symbolptr)</td>
<td>The offset to the symbol in the TLS memory area.</td>
</tr>
</tbody>
</table>

Table 14: Macros for implementing TLS allocation
Managing a multithreaded environment

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:

```c
#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 *symbp)
{
    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(symbp);
    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.
TLS IN THE LINKER CONFIGURATION FILE

Normally, the linker automatically chooses how to initialize static data. If threads are used, the main thread’s TLS memory area must be initialized by plain copying because the initializers are used for each secondary thread’s TLS memory area as well. This is controlled by 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:

<table>
<thead>
<tr>
<th>Object file</th>
<th>Color</th>
<th>Taste</th>
</tr>
</thead>
<tbody>
<tr>
<td>file1</td>
<td>blue</td>
<td>not defined</td>
</tr>
<tr>
<td>file2</td>
<td>red</td>
<td>not defined</td>
</tr>
</tbody>
</table>

Table 15: Example of runtime model attributes
Checking module consistency

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 `mode1` 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", "mode1"
```

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

```
rtmodel "uart", "mode1"
```

Note that key names that start with two underscores are reserved by the compiler. For more information about the syntax, see `rtmodel`, page 338 and the IAR Assembler Reference Guide for ARM, 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 147. The following two are covered in the section Calling convention, page 150.

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 157.

The recommended method for mixing C or C++ and assembler modules is described in Calling assembler routines from C, page 147, and Calling assembler routines from C++, page 149, 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 to 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 to be 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 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:

```c
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):

\[
\text{asm} \ [\text{volatile}]\{ \ \text{string} \ [\text{assembler-interface}] \}
\]

\text{string} can contain one or more valid assembler instructions or data definition assembler directives, separated by \n.

For example:

\[
\text{asm}("\text{label}:\text{nop}\n"
"\text{b label}");
\]

Note that you can define and use local labels in inline assembler instructions.

\text{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

\[
[[\text{symbolic-name}]] \ [\text{modifiers}] \text{constraint} \ (\text{expr})
\]

For example:

\[
\text{int Add}(\text{int term1, int term2})
\{
\text{int sum;}

\text{asm}(\text{"add} \ %0,\ %1,\ %2"
 : "=r" (sum)
 : "r" (term1), "r" (term2));

\text{return sum;}
\}
\]

In this example, the assembler instruction uses one output operand, \text{sum}, two input operands, \text{term1} and \text{term2}, and no clobbered resources.

It is possible to omit any list by leaving it empty. For example:

\[
\text{int matrix}[\text{M}][\text{N}];

\text{void MatrixPreloadRow}(\text{int row})
\{
\text{asm volatile} (\text{"pld} \ [%0]" : : "r" (%matrix[row][0]));
\}
\]
Operand constraints

Operand constraints specify what kind of operand to use, such as a register or an immediate constant. A constraint can also specify what type of register to use, or which constant values are valid.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Uses a general purpose register for the expression: R0-R12, R14 (for ARM and Thumb2) R0-R7 (for Thumb1)</td>
</tr>
<tr>
<td>l</td>
<td>R0-R7 (only valid for Thumb1)</td>
</tr>
<tr>
<td>Rp</td>
<td>Uses a pair of general purpose registers, for example R0, R1</td>
</tr>
<tr>
<td>i</td>
<td>An immediate integer operand with a constant value. Symbolic constants are allowed.</td>
</tr>
<tr>
<td>j</td>
<td>A 16-bit constant suitable for a MOVW instruction (valid for ARM and Thumb2).</td>
</tr>
<tr>
<td>n</td>
<td>An immediate operand, alias for i.</td>
</tr>
<tr>
<td>I</td>
<td>An immediate constant valid for a data processing instruction (for ARM and Thumb2), or a constant in the range 0 to 255 (for Thumb1).</td>
</tr>
<tr>
<td>J</td>
<td>An immediate constant in the range -4095 to 4095 (for ARM and Thumb2), or a constant in the range -255 to -1 (for Thumb1).</td>
</tr>
<tr>
<td>K</td>
<td>An immediate constant that satisfies the I constraint if inverted (for ARM and Thumb2), or a constant that satisfies the I constraint multiplied by any power of 2 (for Thumb1).</td>
</tr>
<tr>
<td>L</td>
<td>An immediate constant that satisfies the I constraint if negated (for ARM and Thumb2), or a constant in the range -7 to 7 (for Thumb1).</td>
</tr>
<tr>
<td>M</td>
<td>An immediate constant that is a multiple of 4 in the range 0 to 1020 (only valid for Thumb1).</td>
</tr>
<tr>
<td>N</td>
<td>An immediate constant in the range 0 to 31 (only valid for Thumb1).</td>
</tr>
<tr>
<td>O</td>
<td>An immediate constant that is a multiple of 4 in the range -508 to 508 (only valid for Thumb1).</td>
</tr>
<tr>
<td>t</td>
<td>An S register.</td>
</tr>
<tr>
<td>w</td>
<td>A D register.</td>
</tr>
<tr>
<td>q</td>
<td>A Q register.</td>
</tr>
<tr>
<td>Dv</td>
<td>A 32-bit floating-point immediate constant for the VMOV.F32 instruction.</td>
</tr>
<tr>
<td>Dy</td>
<td>A 64-bit floating-point immediate constant for the VMOV.F64 instruction.</td>
</tr>
</tbody>
</table>

Table 16: Inline assembler operand constraints
Assembler language interface

Part 1. Using the build tools

Constraint modifiers can be used together with a constraint to modify its meaning. This table lists the supported constraint modifiers:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Write-only operand</td>
</tr>
<tr>
<td>+</td>
<td>Read-write operand</td>
</tr>
<tr>
<td>&amp;</td>
<td>Early clobber output operand which is written to before the instruction has processed all the input operands.</td>
</tr>
</tbody>
</table>

Constraint modifiers

Constraint modifiers can be used together with a constraint to modify its meaning. This table lists the supported constraint modifiers:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2S ... v4Q</td>
<td>A vector of 2, 3, or 4 consecutive S, D, or Q registers. For example, v4Q is a vector of four Q registers. The vectors do not overlap, so the available v4Q register vectors are Q0-Q3, Q4-Q7, Q8-Q11, and Q12-Q15.</td>
</tr>
</tbody>
</table>

Table 16: Inline assembler operand constraints (Continued)

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:

```c
int Add(int term1, int term2)
{
  int sum;

  asm("add %[Rd],%[Rn],%[Rm]"
       : [Rd]"=r"(sum)
       : [Rn]"r" (term1), [Rm]"r" (term2));

  return sum;
}
```

Table 17: Supported constraint modifiers
Operand modifiers

An operand modifier is a single letter between the % and the operand number, which is used for transforming the operand. In the example below, the modifiers L and H are used for accessing the least and most significant 16 bits, respectively, of an immediate operand:

```c
int Mov32()
{
  int a;
  asm("movw %0,%L1 \n"
       "movt %0,%H1 \n" : "=r"(a) : "i"(0x12345678UL));
  return a;
}
```

Some operand modifiers can be combined, in which case each letter will transform the result from the previous modifier. This table describes the transformation performed by each valid modifier:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>The lowest-numbered register of a register pair, or the low 16 bits of an immediate constant.</td>
</tr>
<tr>
<td>H</td>
<td>The highest-numbered register of a register pair, or the high 16 bits of an immediate constant.</td>
</tr>
<tr>
<td>c</td>
<td>For an immediate operand, an integer or symbol address without a preceding # sign. Cannot be transformed by additional operand modifiers.</td>
</tr>
<tr>
<td>B</td>
<td>For an immediate operand, the bitwise inverse of integer or symbol without a preceding # sign. Cannot be transformed by additional operand modifiers.</td>
</tr>
<tr>
<td>Q</td>
<td>The least significant register of a register pair.</td>
</tr>
<tr>
<td>R</td>
<td>The most significant register of a register pair.</td>
</tr>
<tr>
<td>M</td>
<td>For a register or a register pair, the register list suitable for ldmi or stm. Cannot be transformed by additional operand modifiers.</td>
</tr>
<tr>
<td>a</td>
<td>Transforms a register Rn into a memory operand [Rn,#0] suitable for pld.</td>
</tr>
<tr>
<td>b</td>
<td>The low S register part of a D register.</td>
</tr>
<tr>
<td>p</td>
<td>The high S register part of a D register.</td>
</tr>
<tr>
<td>e</td>
<td>The low D register part of a Q register, or the low register in a vector of Neon registers.</td>
</tr>
<tr>
<td>f</td>
<td>The high D register part of a Q register, or the high register in a vector of Neon registers.</td>
</tr>
</tbody>
</table>

Table 18: Operand modifiers and transformations
Assembler language interface

Part 1. Using the build tools

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:

```c
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", ...

---

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>For a (vector of) D or Q registers, the corresponding list of D registers within curly braces. For example, Q0 becomes {D0,D1}. Cannot be transformed by additional operand modifiers.</td>
</tr>
<tr>
<td>y</td>
<td>S register as indexed D register, for example S7 becomes D3[1]. Cannot be transformed by additional operand modifiers.</td>
</tr>
</tbody>
</table>

Table 18: Operand modifiers and transformations
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:

```c
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:

<table>
<thead>
<tr>
<th>Clobber</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R12, R14</td>
<td>for ARM mode and Thumb2</td>
</tr>
<tr>
<td>R0-R7, R12, R14</td>
<td>for Thumb1</td>
</tr>
<tr>
<td>S0-S31, D0-D31, Q0-Q15</td>
<td>Floating-point registers</td>
</tr>
<tr>
<td>cc</td>
<td>The condition flags (N, Z, V, and C)</td>
</tr>
<tr>
<td>memory</td>
<td>To be used if the instructions modify any memory. This will avoid keeping memory values cached in registers across the inline assembler statement.</td>
</tr>
</tbody>
</table>

*Table 19: List of valid clobbers*
THIS IS AN EXAMPLE OF HOW TO USE CLOBBERED MEMORY

```c
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:

  ```c
  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`:

```c
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:

```bash
iccarm 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 157.

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:

```c
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:

```cpp
class MyClass;

extern "C"
{
    void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
    public:
        inline void DoIt(int arg)
        {
            ::DoIt(this, arg);
        }
    }
```

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:

```cpp
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 conventions 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.
The calling convention used by the compiler adheres to the Procedure Call Standard for the ARM architecture, AAPCS, a part of AEABI; see AEABI compliance, page 192. AAPCS is not fully described here. For example, the use of floating-point coprocessor registers when using the VFP calling convention is not covered.

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:

```c
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:

```c
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++:

```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 147.

**Register parameters**

The registers available for passing parameters are R0–R3:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Passed in registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar and floating-point values no larger than 32 bits, and single-precision (32-bits) floating-point values</td>
<td>Passed using the first free register: R0–R3</td>
</tr>
<tr>
<td>long, long, and double-precision (64-bit) values</td>
<td>Passed in first available register pair: R0:R1, or R2:R3</td>
</tr>
</tbody>
</table>

*Table 20: 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:

![Stack image after the function call](image)

**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`.

<table>
<thead>
<tr>
<th>Return values</th>
<th>Passed in registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar and structure return values no larger than 32 bits, and single-precision (32-bit) floating-point return values</td>
<td><code>R0</code></td>
</tr>
<tr>
<td>The memory address of a structure return value larger than 32 bits</td>
<td><code>R0</code></td>
</tr>
<tr>
<td><code>long long</code> and double-precision (64-bit) return values</td>
<td><code>R0:R1</code></td>
</tr>
</tbody>
</table>

*Table 21: 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:

```assembly
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:

```c
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:

```assembly
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:

```c
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`.

```c
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:

```c
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 IAR Assembler Reference Guide for ARM.

**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:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFA R13</td>
<td>The call frames of the stack</td>
</tr>
<tr>
<td>R0–R12</td>
<td>Processor general-purpose 32-bit registers</td>
</tr>
<tr>
<td>R13</td>
<td>Stack pointer, SP</td>
</tr>
<tr>
<td>R14</td>
<td>Link register, LR</td>
</tr>
<tr>
<td>D0–D31</td>
<td>Vector Floating Point (VFP) 64-bit coprocessor register</td>
</tr>
<tr>
<td>CPSR</td>
<td>Current program status register</td>
</tr>
<tr>
<td>SPSR</td>
<td>Saved program status register</td>
</tr>
</tbody>
</table>

*Table 22: Call frame information resources defined in a names block*
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:

   ```c
   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:

```assembly
NAME Cfi

RTMODEL "__SystemLibrary", "DLib"

EXTERN F

PUBLIC cfiExample

CFI Names cfiNames0
CFI StackFrame CFA R13 DATA
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)
```

Part 1. Using the build tools 159
Call frame information

```assembly
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_NAMES_BLOCK`, `CFI_COMMON_ARM`, and `CFI_COMMON_Thumb`, which declare a typical names block and a typical common block. 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 --c99 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 defined immediately after the keyword 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 139.
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 163. 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 137. 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 391.

**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:

<table>
<thead>
<tr>
<th>Command line</th>
<th>IDE*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--strict</code></td>
<td>Strict</td>
<td>All IAR C language extensions are disabled; errors are issued for anything that is not part of Standard C.</td>
</tr>
<tr>
<td>None</td>
<td>Standard</td>
<td>All extensions to Standard C are enabled, but no extensions for embedded systems programming. For information about extensions, see <em>IAR C language extensions</em>, page 163.</td>
</tr>
<tr>
<td><code>-e</code></td>
<td>Standard with IAR extensions</td>
<td>All IAR C language extensions are enabled.</td>
</tr>
</tbody>
</table>

* 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 166.
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 201, and location, page 333.

- Alignment control
  Each data type has its own alignment; for more information, see Alignment, page 297. 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 199.

- 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 300.

- 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 \ldots parameter, the comma is then deleted in the ", ##__VA_ARGS__" macro definition. According to Standard C, the \ldots 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:

\begin{verbatim}
__section_begin
__section_end
__section_size
\end{verbatim}

\textbf{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:

\begin{verbatim}
void * __section_begin(char const * section)
void * __section_end(char const * section)
size_t * __section_size(char const * section)
\end{verbatim}

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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>__section_begin(sec)</td>
<td>sec$$Base</td>
</tr>
<tr>
<td>__section_end(sec)</td>
<td>sec$$Limit</td>
</tr>
<tr>
<td>__section_size(sec)</td>
<td>sec$$Length</td>
</tr>
</tbody>
</table>

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 *

```c
#pragma section="MYSECTION"
...
ssection_start_address = __section_begin("MYSECTION");
```

See also section, page 340, and location, page 333.

**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 306.

- **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.
IAR C language extensions

- 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:
  ```c
  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.
  ```c
  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:
  ```c
  "void func(char)"
  ```
  These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see -e, page 242.
● 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
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++.

- 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.
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 252.

**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 254.

**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 237.

To compile files written in Embedded C++, you must use the --ec++ compiler option. See --ec++, page 243.

To take advantage of Extended Embedded C++ features in your source code, you must use the --eec++ compiler option. See --eec++, page 243.

To enable EC++, or 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**

```cpp
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();

    // 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 113.

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 compiler supports templates with the syntax and semantics as defined by Standard C++. However, note that the STL (standard template library) delivered with the product is tailored for Extended EC++, see Extended Embedded C++, page 172.

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.
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:
  ```cpp
  class B;
  class A
  {
    friend B;       // Possible when using IAR language extensions
                    // According to the standard
    friend class B; // According to the standard
  };
  ```

- Constants of a scalar type can be defined within classes, for example:
  ```cpp
  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:
  ```cpp
  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:
  ```cpp
  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:
  ```cpp
  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:

```cpp
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:

```cpp
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:

```cpp
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
};
```
EC++ and C++ language extensions

**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 440.

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 89, and Saving stack space and RAM memory, page 211.

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 150 and more specifically Special registers, page 152 and Stack parameters and layout, page 153.

EXCEPTION STACK

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:

<table>
<thead>
<tr>
<th>Processor mode</th>
<th>Proposed stack section name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor</td>
<td>SVC_STACK</td>
<td>Operation system stack.</td>
</tr>
<tr>
<td>IRQ</td>
<td>IRQ_STACK</td>
<td>Stack for general-purpose (IRQ) interrupt handlers.</td>
</tr>
<tr>
<td>FIQ</td>
<td>FIQ_STACK</td>
<td>Stack for high-speed (FIQ) interrupt handlers.</td>
</tr>
<tr>
<td>Undefined</td>
<td>UND_STACK</td>
<td>Stack for undefined instruction interrupts. Supports software emulation of hardware coprocessors and instruction set extensions.</td>
</tr>
</tbody>
</table>

Table 25: Exception stacks
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 90.

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:

```bash
--define_symbol NrOfElements=10
--config_def HeapSize=1024
```

The linker configuration file can look like this:

```plaintext
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:

```c
#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.

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`):

```plaintext
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`:

```plaintext
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:

```c
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:

```c
/* 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 440.
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 294 and --no_vfe, page 289.

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 192.

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 194.

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>
#define _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><l|b>[f]_math.a
```

where `<0|3|4>` selects the Cortex–M variant, `<l|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)**

```
iccarm -I $EW_DIR$/arm\CMSIS\Include
```

**With the DSP library, for Cortex-M4, little-endian, and with FPU**

```
iccarm --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_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`. 
Selecting data types

- Bitfields and packed structures generate large and slow code.
- Using floating-point types on a microprocessor without a math co-processor 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, the calculation is performed using double precision, because one of the operands—the constant `1.0`—is of the type `double`:

```c
float Test(float a, float b)
{
    return a + 1.0 + b;
}
```

To treat a floating-point constant as a `float` rather than as a `double`, add the suffix `f` to it, for example:

```c
float Test(float a, float b)
{
    return a + 1.0f + b;
}
```

For more information about floating-point types, see *Floating-point types*, page 304.
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 297.

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 337.

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 242, 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:

```c
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:

```c
__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 \( \text{IOPORT} \) at address \( \text{0x1000} \). The I/O register has 2 bits declared, \( \text{Way} \) and \( \text{Out} \). 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.

  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 263.
At compile time, data and functions are placed in different sections, see *Modules and sections*, page 70. 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 73.

**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:

```c
__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.

```c
#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:

```c
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:

```c
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:

```c
/* 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.
Controlling data and function placement in memory

**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.

```c
__no_init int alpha @ "MY_NOINIT"; /* OK */

#pragma location="MY_CONSTANTS"
const int beta = 42; /* OK */

const int gamma @ "MY_CONSTANTS" = 17; /* OK */
int theta @ "MY_ZEROS"; /* OK */
int phi @ "MY_INITED" = 4711; /* OK */
```

The linker will in the zero and initialized cases arrange for the correct type of initialization for the variable. When placing a `__no_init` variable in a user-defined section, you must add a pattern that matches that section to your `do not initialize` directive in the linker configuration file. For initialized variables, you can disable the automatic initialization by using the `initialize manually` directive.

### Examples of placing functions in named sections

```c
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.
Controlling compiler optimizations

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 335, 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-source code 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 250.

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 241.
OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. This table lists optimizations that are typically performed on each level:

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (Best debug support)</td>
<td>Variables live through their entire scope</td>
</tr>
<tr>
<td>Low</td>
<td>Same as above but variables only live for as long as they are needed, not necessarily through their entire scope</td>
</tr>
<tr>
<td>Medium</td>
<td>Same as above, and: Live-dead analysis and optimization</td>
</tr>
<tr>
<td></td>
<td>Dead code elimination</td>
</tr>
<tr>
<td></td>
<td>Redundant label elimination</td>
</tr>
<tr>
<td></td>
<td>Redundant branch elimination</td>
</tr>
<tr>
<td></td>
<td>Code hoisting</td>
</tr>
<tr>
<td></td>
<td>Peephole optimization</td>
</tr>
<tr>
<td></td>
<td>Some register content analysis and optimization</td>
</tr>
<tr>
<td></td>
<td>Static clustering</td>
</tr>
<tr>
<td></td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>High (Balanced)</td>
<td>Same as above, and: Instruction scheduling</td>
</tr>
<tr>
<td></td>
<td>Cross jumping</td>
</tr>
<tr>
<td></td>
<td>Advanced register content analysis and optimization</td>
</tr>
<tr>
<td></td>
<td>Loop unrolling</td>
</tr>
<tr>
<td></td>
<td>Function inlining</td>
</tr>
<tr>
<td></td>
<td>Code motion</td>
</tr>
<tr>
<td></td>
<td>Type-based alias analysis</td>
</tr>
</tbody>
</table>

Table 26: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see Fine-tuning enabled transformations, page 208.

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.

If you use the optimization level High speed, the \texttt{--no\_size\_constraints} compiler option relaxes the normal restrictions for code size expansion and enables more aggressive optimizations.

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 \texttt{--no\_inline}, alternatively its equivalent in the IDE Function inlining, or the \texttt{#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.

\textbf{Note:} This option has no effect at optimization levels None and Low.

For more information about the command line option, see \texttt{--no\_cse}, page 252.

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 257.

**Function inlining**

Function inlining means that a 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 normally reduces execution time, but might increase the code size.

For more information, see *Inlining functions*, page 65.

**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 251.

**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.
Controlling compiler optimizations

**Note:** This option has no effect at optimization levels None, Low, and Medium.
For more information about the command line option, see `--no_thaa`, page 256.

**Example**

```c
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 251.

**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 255.
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 209. 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 206.
- Avoid using inline assembler without operands and clobbered resources. Instead, use SFRs or intrinsic functions if available. Otherwise, use inline assembler with operands and clobbered resources or write a separate module in assembler language. For more information, see Mixing C and assembler, page 137.

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 valid C, however it is strongly recommended to use the prototyped style, and provide a prototype declaration for each public function in a header that is included both in the compilation unit defining the function and in all compilation units using it.

The compiler will not perform type checking on parameters passed to functions declared using K&R style. Using prototype declarations will also result in more efficient code in some cases, as there is no need for type promotion for these functions.

To make the compiler require that all function definitions use the prototyped style, and that all public functions have been declared before being defined, use the Project>Options>C/C++ Compiler>Language 1>Require prototypes compiler option ("--require_prototypes").

Prototyped style

In prototyped function declarations, the type for each parameter must be specified.

```c
int Test(char, int); /* Declaration */
```

```c
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:

```c
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:

```c
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 0xFFFFFF7F. 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 309.

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:

```c
__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. */

```c
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:

```c
static unsigned long get_APSR( void )
{
    unsigned long value;
    asm volatile("MRS %0, APSR" : "=r"(value));
    return value;
}
static void set_APSR( unsigned long value)
{
    asm volatile("MSR APSR, %0" :: "r"(value));
}
```

The general purpose register is used for getting and setting the value of the special purpose register APSR. 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 139. An alternative to inline assembler can be to use an intrinsic function instead. See Summary of intrinsic functions, page 345.

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 more information, see __no_init, page 319. Note that to use this keyword, language extensions must be enabled; see -e, page 242. For more information, see also object_attribute, page 335.
Facilitating good code generation
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:

```
iccarm [options] [sourcefile] [options]
```

For example, when compiling the source file `prog.c`, use this command to generate an object file with debug information:

```
iccarm 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 219.

- Via environment variables
  
  The compiler and linker automatically append the value of the environment variables to every command line; see `Environment variables`, page 220.

- Via a text file, using the `-f` option; see `-f`, page 245.

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:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_INCLUDE</td>
<td>Specifies directories to search for include files; for example: <code>C_INCLUDE=c:\program files\iar systems\embedded workbench 6.n\arm\inc; c:\headers</code></td>
</tr>
<tr>
<td>QCCARM</td>
<td>Specifies command line options; for example: <code>QCCARM=-1A asm.lst</code></td>
</tr>
</tbody>
</table>

Table 27: Compiler environment variables
This environment variable can be used with ILINK:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILINKARM_CMD_LINE</td>
<td>Specifies command line options; for example: ILINKARM_CMD_LINE=--config full.icf --silent</td>
</tr>
</tbody>
</table>

Table 28: 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:

  ```c
  #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 247.

  2. The directories specified using the `C_INCLUDE` environment variable, if any; see `Environment variables`, page 220.

  3. The automatically set up library system include directories. See `--dlib_config`, page 242.

- If the compiler encounters the name of an `#include` file in double quotes, for example:

  ```c
  #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:

```c
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:

```c
iccarm ..\src\src.c -I..\include -I..\debugconfig```

---

Part 2. Reference information  221

AFE1_AFE2-1:1
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:

- dir\include: Current file is src.h.
- dir\src: File including current file (src.c).
- dir\include: As specified with the first -I option.
- dir\debugconfig: As specified with the second -I 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 381.

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 248. 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 224.

- 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 223.

- 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compilation or linking successful, but there might have been warnings.</td>
</tr>
<tr>
<td>1</td>
<td>Warnings were produced and the option \texttt{--warnings affect exit code} was used.</td>
</tr>
<tr>
<td>2</td>
<td>Errors occurred.</td>
</tr>
<tr>
<td>3</td>
<td>Fatal errors occurred, making the tool abort.</td>
</tr>
<tr>
<td>4</td>
<td>Internal errors occurred, making the tool abort.</td>
</tr>
</tbody>
</table>

*Table 29: 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...
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:

- **filename**: The name of the source file in which the issue was encountered
- **linenumber**: The line number at which the compiler detected the issue
- **level**: The level of seriousness of the issue
- **tag**: A unique tag that identifies the diagnostic message
- **message**: 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.

---

map, see `--map`, page 285. 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 224.

- **Error return codes**
  ILINK returns status information to the operating system which can be tested in a batch file, see *Error return codes*, page 223.

- **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.
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:

\textit{level}[\textit{tag}]: \textit{message}

with these elements:

\begin{itemize}
  \item \textit{level} The level of seriousness of the issue
  \item \textit{tag} A unique tag that identifies the diagnostic message
  \item \textit{message} An explanation, possibly several lines long
\end{itemize}

Diagnostic messages are displayed on the screen, as well as printed in the optional map file.

Use the option \texttt{--diagnostics\_tables} to list all possible linker diagnostic messages.

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

\textbf{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 \texttt{--remarks}, page 262.

\textbf{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 \texttt{--no\_warnings}, see page 258.

\textbf{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.

\textbf{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 229, 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 220.

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
Part 2. Reference information

Compiler options

- 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:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--aapcs</td>
<td>Specifies the calling convention</td>
</tr>
<tr>
<td>--aeabi</td>
<td>Enables AEABI-compliant code generation</td>
</tr>
<tr>
<td>--align_sp_on_irq</td>
<td>Generates code to align SP on entry to __irq</td>
</tr>
<tr>
<td></td>
<td>functions</td>
</tr>
<tr>
<td>--arm</td>
<td>Sets the default function mode to ARM</td>
</tr>
</tbody>
</table>

Table 30: Compiler options summary
### Summary of compiler options

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--c89</td>
<td>Specifies the C89 dialect</td>
</tr>
<tr>
<td>--char_is_signed</td>
<td>Treats char as signed</td>
</tr>
<tr>
<td>--char_is_unsigned</td>
<td>Treats char as unsigned</td>
</tr>
<tr>
<td>--cpu</td>
<td>Specifies a processor variant</td>
</tr>
<tr>
<td>--c++</td>
<td>Specifies Standard C++</td>
</tr>
<tr>
<td>-D</td>
<td>Defines preprocessor symbols</td>
</tr>
<tr>
<td>--debug</td>
<td>Generates debug information</td>
</tr>
<tr>
<td>--dependencies</td>
<td>Lists file dependencies</td>
</tr>
<tr>
<td>--diag_error</td>
<td>Treats these as errors</td>
</tr>
<tr>
<td>--diag_remark</td>
<td>Treats these as remarks</td>
</tr>
<tr>
<td>--diag_suppress</td>
<td>Suppresses these diagnostics</td>
</tr>
<tr>
<td>--diag_warning</td>
<td>Treats these as warnings</td>
</tr>
<tr>
<td>--diagnostics_tables</td>
<td>Lists all diagnostic messages</td>
</tr>
<tr>
<td>--discard_unused_publics</td>
<td>Discards unused public symbols</td>
</tr>
<tr>
<td>--dlib_config</td>
<td>Uses the system include files for the DLIB library and determines which configuration of the library to use</td>
</tr>
<tr>
<td>-e</td>
<td>Enables language extensions</td>
</tr>
<tr>
<td>--ec++</td>
<td>Specifies Embedded C++</td>
</tr>
<tr>
<td>--eec++</td>
<td>Specifies Extended Embedded C++</td>
</tr>
<tr>
<td>--enable_hardware_workaround</td>
<td>Enables a specific hardware workaround</td>
</tr>
<tr>
<td>--enable_multibytes</td>
<td>Enables support for multibyte characters in source files</td>
</tr>
<tr>
<td>--endian</td>
<td>Specifies the byte order of the generated code and data</td>
</tr>
<tr>
<td>--enum_is_int</td>
<td>Sets the minimum size on enumeration types</td>
</tr>
<tr>
<td>--error_limit</td>
<td>Specifies the allowed number of errors before compilation stops</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line</td>
</tr>
<tr>
<td>--fpu</td>
<td>Selects the type of floating-point unit</td>
</tr>
<tr>
<td>--guard_calls</td>
<td>Enables guards for function static variable initialization</td>
</tr>
<tr>
<td>--header_context</td>
<td>Lists all referred source files and header files</td>
</tr>
</tbody>
</table>

*Table 30: Compiler options summary (Continued)*
### Compiler options

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-I</td>
<td>Specifies include file path</td>
</tr>
<tr>
<td>--interwork</td>
<td>Generates interworking code</td>
</tr>
<tr>
<td>-l</td>
<td>Creates a list file</td>
</tr>
<tr>
<td>--legacy</td>
<td>Generates object code linkable with older tool chains</td>
</tr>
<tr>
<td>--lock_regs</td>
<td>Prevents the compiler from using specified registers</td>
</tr>
<tr>
<td>--macro_positions_in</td>
<td>Obtain positions inside macros in diagnostic messages</td>
</tr>
<tr>
<td>_diagnostics</td>
<td></td>
</tr>
<tr>
<td>--mfc</td>
<td>Enables multi-file compilation</td>
</tr>
<tr>
<td>--misrac</td>
<td>Enables error messages specific to MISRA-C:1998. This option is a synonym of --misrac1998 and is only available for backwards compatibility.</td>
</tr>
<tr>
<td>--no_clustering</td>
<td>Disables static clustering optimizations</td>
</tr>
<tr>
<td>--no_code_motion</td>
<td>Disables code motion optimization</td>
</tr>
<tr>
<td>--no_cse</td>
<td>Disables common subexpression elimination</td>
</tr>
<tr>
<td>--no_const_align</td>
<td>Disables the alignment optimization for constants.</td>
</tr>
<tr>
<td>--no_exceptions</td>
<td>Disables C++ exception support</td>
</tr>
<tr>
<td>--no.fragments</td>
<td>Disables section fragment handling</td>
</tr>
<tr>
<td>--no_inline</td>
<td>Disables function inlining</td>
</tr>
<tr>
<td>--no_path_in_file_macros</td>
<td>Removes the path from the return value of the symbols <strong>FILE</strong> and <strong>BASE_FILE</strong></td>
</tr>
<tr>
<td>--no_rtti</td>
<td>Disables C++ RTTI support</td>
</tr>
<tr>
<td>--no_rw_dynamic_init</td>
<td>Disables runtime initialization of static C variables.</td>
</tr>
<tr>
<td>--no_scheduling</td>
<td>Disables the instruction scheduler</td>
</tr>
</tbody>
</table>

Table 30: Compiler options summary (Continued)
Summary of compiler options

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--no_size_constraints</td>
<td>Relaxes the normal restrictions for code size expansion when optimizing for speed.</td>
</tr>
<tr>
<td>--no_static_destruction</td>
<td>Disables destruction of C++ static variables at program exit</td>
</tr>
<tr>
<td>--no_system_include</td>
<td>Disables the automatic search for system include files</td>
</tr>
<tr>
<td>--no_ttas</td>
<td>Disables type-based alias analysis</td>
</tr>
<tr>
<td>--no_typedefs_in_diagnostics</td>
<td>Disables the use of typedef names in diagnostics</td>
</tr>
<tr>
<td>--no_unaligned_access</td>
<td>Avoids unaligned accesses</td>
</tr>
<tr>
<td>--no_unroll</td>
<td>Disables loop unrolling</td>
</tr>
<tr>
<td>--no_warnings</td>
<td>Disables all warnings</td>
</tr>
<tr>
<td>--no_wrap_diagnostics</td>
<td>Disables wrapping of diagnostic messages</td>
</tr>
<tr>
<td>-O</td>
<td>Sets the optimization level</td>
</tr>
<tr>
<td>-o</td>
<td>Sets the object filename. Alias for --output.</td>
</tr>
<tr>
<td>--only_stdout</td>
<td>Uses standard output only</td>
</tr>
<tr>
<td>--output</td>
<td>Sets the object filename</td>
</tr>
<tr>
<td>--predef_macros</td>
<td>Lists the predefined symbols.</td>
</tr>
<tr>
<td>--preinclude</td>
<td>Includes an include file before reading the source file</td>
</tr>
<tr>
<td>--preprocess</td>
<td>Generates preprocessor output</td>
</tr>
<tr>
<td>--public_equ</td>
<td>Defines a global named assembler label</td>
</tr>
<tr>
<td>-r</td>
<td>Generates debug information. Alias for --debug.</td>
</tr>
<tr>
<td>--relaxed_fp</td>
<td>Relaxes the rules for optimizing floating-point expressions</td>
</tr>
<tr>
<td>--remarks</td>
<td>Enables remarks</td>
</tr>
<tr>
<td>--require_prototypes</td>
<td>Verifies that functions are declared before they are defined</td>
</tr>
<tr>
<td>--ropi</td>
<td>Generates code that uses PC-relative references to address code and read-only data.</td>
</tr>
<tr>
<td>--rwpi</td>
<td>Generates code that uses an offset from the static base register to address-writable data.</td>
</tr>
<tr>
<td>--section</td>
<td>Changes a section name</td>
</tr>
<tr>
<td>--separate_cluster_for_initialized_variables</td>
<td>Separates initialized and non-initialized variables</td>
</tr>
</tbody>
</table>

Table 30: 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aapcs={std</td>
<td>vfp}</td>
</tr>
</tbody>
</table>

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>std</td>
<td>Processor registers are used for floating-point parameters and return values in function calls according to standard AAPCS. std is the default when the software FPU is selected.</td>
</tr>
<tr>
<td>vfp</td>
<td>VFP registers are used for floating-point parameters and return values. The generated code is not compatible with AEABI code. vfp is the default when a VFP unit is used.</td>
</tr>
</tbody>
</table>

**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**  
--aeabi

**Description**  
Use this option to generate AEABI-compliant object code. Note that this option must be used together with the --guard_calls option.

**See also**  
AEABI compliance, page 192 and --guard_calls, page 247.

To set this option, use Project>Options>C/C++ Compiler>Extra Options.

### --align_sp_on_irq

**Syntax**  
--align_sp_on_irq

**Description**  
Use this option to align the stack pointer (SP) on entry to __irq 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**  
__irq, page 318.

To set this option, use Project>Options>C/C++ Compiler>Extra Options.

### --arm

**Syntax**  
--arm

**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 --cpu_mode=arm option.

**See also**  

Project>Options>C/C++ Compiler>Code>Processor mode>Arm
**--c89**

Syntax: `--c89`

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: [C language overview](#), page 161.

---

**--char_is_signed**

Syntax: `--char_is_signed`

Description: By default, the compiler interprets the plain `char` type as unsigned. Use this option to make the compiler interpret the plain `char` 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 `--char_is_signed` option and cannot be used with code that is compiled with this option.

---

**--char_is_unsigned**

Syntax: `--char_is_unsigned`

Description: Use this option to make the compiler interpret the plain `char` type as unsigned. This is the default interpretation of the plain `char` type.

---

**--cpu**

Syntax: `--cpu=core`

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>core</code></td>
<td>Specifies a specific processor variant</td>
</tr>
</tbody>
</table>

---
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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Processor Type</th>
<th>Exception</th>
<th>Flags</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM7TDMI</td>
<td>ARM10E</td>
<td>Cortex-M0+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM7TDMI-S</td>
<td>ARM1020E</td>
<td>Cortex-M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM710T</td>
<td>ARM1022E</td>
<td>Cortex-M1*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM720T</td>
<td>ARM1026BJ-S</td>
<td>Cortex-M3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM740T</td>
<td>ARM1136J</td>
<td>Cortex-M4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM7BJ-S</td>
<td>ARM1136J-S</td>
<td>Cortex-M4F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM9TDMI</td>
<td>ARM1136JF</td>
<td>Cortex-R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM920T</td>
<td>ARM1136JF-S</td>
<td>Cortex-R4F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM922T</td>
<td>ARM1176J</td>
<td>Cortex-R5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(alias for ARM1176JZ)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM940T</td>
<td>ARM1176J-S</td>
<td>Cortex-R5F</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(alias for ARM1176JZ-S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM9E</td>
<td>ARM1176JF</td>
<td>Cortex-R7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(alias for ARM1176JZF)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM9E-S</td>
<td>ARM1176JF-S</td>
<td>Cortex-R7F</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(alias for ARM1176JZF-S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM926BJ-S</td>
<td>Cortex-A5</td>
<td>XScale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM966E-S</td>
<td>Cortex-A7</td>
<td>XScale-IR7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM968E-S</td>
<td>Cortex-A15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM946E-S</td>
<td>Cortex-M0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Cortex-M1 with Operating System extension.

See also

Process variant, page 48

Project>Options>General Options>Target>Processor configuration
Compiler options

--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
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
Project>Options>C/C++ Compiler>Language 1>C++
and
Project>Options>C/C++ Compiler>Language 1>C++ dialect>C++

-D
Syntax
-D symbol[=value]
Parameters
symbol The name of the preprocessor symbol
value 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 `-D` has the same effect as a `#define` statement at the top of the source file:

```
-Dsymbol1
```

is equivalent to:
```
#define symbol1 1
```
To get the equivalence of:
```
#define FOO
```
specify the `=` sign but nothing after, for example:
```
-DFOO=
```

**Project>Options>C/C++ Compiler>Preprocessor>Defined symbols**

---

**--debug, -r**

**Syntax**
```
--debug
-r
```

**Description**
Use the `--debug` or `-r` 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 228.

**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`. 

---

**IAR C/C++ Development Guide**

238 Compiling and Linking for ARM
Example

If \texttt{\--dependencies} or \texttt{\--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:

\begin{verbatim}
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
\end{verbatim}

If \texttt{\--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:

\begin{verbatim}
foo.o: c:\iar\product\include\stdio.h
foo.o: d:\myproject\include\foo.h
\end{verbatim}

An example of using \texttt{\--dependencies} with a popular make utility, such as gmake (GNU make):

1. Set up the rule for compiling files to be something like:

\begin{verbatim}
%.o : %.c
 $(ICC) $(ICCFLAGS) $< \--dependencies=m $*.d
\end{verbatim}

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:

\begin{verbatim}
-include $(sources:.c=.d)
\end{verbatim}

Because of the dash (\texttt{-}) it works the first time, when the .d files do not yet exist.

This option is not available in the IDE.

\textbf{\--diag\_error}

Syntax

\texttt{\--diag\_error=tag[,,tag,...]}

Parameters

\texttt{tag} \hspace{1cm} 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
Descriptions of compiler options

--diag_remark
Syntax
--diag_remark=tag[, tag,...]
Parameters
tag 
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 --remarks option to display them.

--diag_suppress
Syntax
--diag_suppress=tag[, tag,...]
Parameters
tag 
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.

--diag_warning
Syntax
--diag_warning=tag[, tag,...]
Parameters
tag 
Description
The number of a diagnostic message, for example the message number Pe177
The number of a diagnostic message, for example the message number Pe117
The number of a diagnostic message, for example the message number Pe826
**Compiler options**

---

**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 228.

**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 250 and *Multi-file compilation units*, page 206.

*Project > Options > C/C++ Compiler > Discard unused publics*
Descriptions of compiler options

**--dlib_config**

**Syntax**

```
--dlib_config filename.h|config
```

**Parameters**

- `filename` A DLIB configuration header file. For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 228.
- `config` 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 101.

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 112.

To set related options, choose:

Project>Options>General Options>Library Configuration

**-e**

**Syntax**

```
-e
```

**Description**

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.

IAR C/C++ Development Guide

242 Compiling and Linking for ARM
Note: The -e option and the --strict option cannot be used at the same time.

See also
Enabling language extensions, page 163.
Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions
Note: By default, this option is selected in the IDE.

--ec++
Syntax
--ec++
Description
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++.
Project>Options>C/C++ Compiler>Language 1>C++
and
Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++

--eec++
Syntax
--eec++
Description
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++.
See also
Extended Embedded C++, page 172.
Project>Options>C/C++ Compiler>Language 1>C++
and
Project>Options>C/C++ Compiler>Language 1>C++ dialect>Extended Embedded C++
**--enable_hardware_workaround**

**Syntax**

```
--enable_hardware_workaround=waid[,waid...]
```

**Parameters**

- `waid` 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**

```
--enable_multibytes
```

**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**

```
--endian={big|b|little|l}
```

**Parameters**

- `big, b` Specifies big-endian as the default byte order
- `little, l` (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 298."

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 299."

To set this option, use Project>Options>C/C++ Compiler>Extra Options.

**--error_limit**

**Syntax**

```
--error_limit=n
```

**Parameters**

```
n
```

The number of errors before the compiler stops the compilation. **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 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 228."
Descriptions of compiler 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.

**Note:** This option requires a threaded C++ environment, which is not supported in the IAR C/C++ Compiler for ARM.

See also *Managing a multithreaded environment*, page 130.

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 path

**Parameters**

path  

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.
--- 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.

--- interwork, page 318.

--- interwork

**Project>Options>General Options>Target>Generate interwork code**

---

**-l**

**Syntax**

```
-l[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 `-la`, 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 `-lA`, 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
Compiler options

--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 generate code that should be linked with the linker in RVCT3.0.

Description
Use this option to generate code compatible with the specified toolchain.

To set this option, use *Project>Options>C/C++ Compiler>Extra Options*.

--lock_regs

Syntax
--lock_regs=register

Parameters
registers A comma-separated list of register names and register intervals to be locked, in the range R4–R11.

Description
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 228.

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

Part 2. Reference information 249
Descriptions of compiler options

---

**Description**

Use this option to prevent the compiler from generating code that uses the specified registers.

**Example**

```
--lock_reg=R4
--lock_reg=R8-R11
--lock_reg=R4,R8-R11
```

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

---

**--macro_positions_in_diagnostics**

**Syntax**

```
--macro_positions_in_diagnostics
```

**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**

```
--mfc
```

**Description**

Use this option to enable *multi-file compilation*. 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 **-o** compiler option and specify a certain output file.

**Example**

```
iccarm myfile1.c myfile2.c myfile3.c --mfc
```

**See also**

```
```

---

**Project>Options>C/C++ Compiler>Multi-file compilation**
--no_clustering

Syntax
--no_clustering

Description
Use this option to disable static clustering optimizations.

Note: This option has no effect at optimization levels below Medium.

See also

--no_code_motion

Syntax
--no_code_motion

Description
Use this option to disable code motion optimizations.

Note: This option has no effect at optimization levels below Medium.

See also
Code motion, page 209.

--no_const_align

Syntax
--no_const_align

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 `const` 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 297

To set this option, use Project>Options>C/C++ Compiler>Extra Options.
Descriptions of compiler 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 208.

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 174.

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.
Part 2. Reference information

Compiler options

See also

Keeping symbols and sections, page 89.

To set this option, use Project>Options>C/C++ CompilerLinker>Extra Options

--no_inline

Syntax

--no_inline

Description

Use this option to disable function inlining.

See also

Inlining functions, page 65.

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 297

To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--no_mem_idioms

Syntax

--no_mem_idioms

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

```
--no_path_in_file_macros
```

Description
Use this option to exclude the path from the return value of the predefined preprocessor symbols __FILE__ and __BASE_FILE__.

See also

Description of predefined preprocessor symbols, page 382.

This option is not available in the IDE.

**--no_rtti**

Syntax

```
--no_rtti
```

Description
Use this option to disable the runtime type information (RTTI) support in the C++ language. RTTI statements like `dynamic_cast` and `typeid` will generate an error message. This option is only valid when used together with the --c++ compiler option.

See also

Using C++, page 171.

```
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 262 and --rwpi, page 263
```

```
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**


---no_size_constraints

**Syntax**

--no_size_constraints

**Description**

Use this option to optimize for speed but relax the normal restrictions for code size expansion.

**Note:** This option has no effect unless used with -Ohs.

**See also**

Speed versus size, page 208.

---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 90.

To set this option, use Project>Options>C/C++ Compiler>Extra Options.
--no_system_include

Syntax
--no_system_include

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 -I compiler option.

See also
--dlib_config, page 242, and --system_include_dir, page 265.

Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories

--no_tbaa

Syntax
--no_tbaa

Description
Use this option to disable type-based alias analysis.

Note: This option has no effect at optimization levels below High.

See also
Type-based alias analysis, page 209.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis

--no_typedefs_in_diagnostics

Syntax
--no_typedefs_in_diagnostics

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
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";

will give an error message like this:

Error[Pe144]: a value of type 'char **' cannot be used to initialize an entity of type 'MyPtr'
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 297.

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 208.

To set this option, use `Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling`
Descriptions of compiler options

--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 205.

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 228.

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 228.

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.
Descriptions of compiler options

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 228.

**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 `#line` directives

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 228.

**Description**

Use this option to generate preprocessed output to a named file.

*Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file*
Compiler options

---

### --public_equ

**Syntax**  
`--public_equ symbol [=value]`

**Parameters**

- `symbol`  
The name of the assembler symbol to be defined
- `value`  
An optional value of the defined assembler symbol

**Description**  
This option is equivalent to defining a label in assembler language using the `EQU` directive and exporting it using the `PUBLIC` directive. This option can be used more than once on the command line.

---

### --relaxed_fp

**Syntax**  
`--relaxed_fp`

**Description**  
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:

- 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.

Note that performing the calculation in single precision instead of double precision might cause a loss of accuracy.

**Example**

```c
float F(float a, float b)
{
    return a + b * 3.0;
}
```

The C standard states that `3.0` in this example has the type `double` and therefore the whole expression should be evaluated in `double` precision. However, when the `--relaxed_fp` option is used, `3.0` will be converted to `float` and the whole expression can be evaluated in `float` precision.

To set related options, choose:

*Project>*Options>*C/C++ Compiler>*Language 2>*Floating-point semantics*
Descriptions of compiler options

--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 Severity levels, page 225.

Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes
Syntax     --require_prototypes
Description 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:

- 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 Use this option to make the compiler generate code that uses PC-relative references to address code and read-only data.

When this option is used, these limitations apply:
- C++ constructions cannot be used
- The object attribute __ramfunc 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_rwlock_dynamic_init, page 254, and Description of predefined preprocessor symbols, page 382 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 addresses at runtime.

See also
--no_rwlock_dynamic_init, page 254, and Description of predefined preprocessor symbols, page 382 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.
Descriptions of compiler options

**Example**

To place functions in the section `MyText`, use:

```
--section .text=MyText
```

**See also**

*Controlling data and function placement in memory*, page 201.

---

**--separate_cluster_for_initialized_variables**

**Syntax**

```
--separate_cluster_for_initialized_variables
```

**Description**

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.

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.

**See also**


To set this option, use `Project>Options>C/C++ Compiler>Extra Options`.

---

**--silent**

**Syntax**

```
--silent
```

**Description**

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).

This option does not affect the display of error and warning messages.

This option is not available in the IDE.
Compiler options

--strict
Syntax
--strict
Description
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++.

**Note:** The `-e` option and the `--strict` option cannot be used at the same time.

See also
Enabling language extensions, page 163.

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 228.

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 242, and `--no_system_include`, page 256.

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.
Descriptions of compiler options

See also  


Project>Options>C/C++ Compiler>Code>Processor mode>Thumb

--use_c++_inline
Syntax  

--use_c++_inline
Description  

Standard C uses slightly different semantics for the inline keyword than C++ does. Use this option if you want C++ semantics when you are using C.

See also  

Inlining functions, page 65

Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

--use_unix_directory_separators
Syntax  

--use_unix_directory_separators
Description  

Use this option to make DWARF debug information use / (instead of \) as directory separators in file paths.

This option can be useful if you have a debugger that requires directory separators in UNIX style.

To set this option, use Project>Options>C/C++ Compiler>Extra Options.

--vla
Syntax  

--vla
Description  

Use this option to enable support for C99 variable length arrays. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the --c89 compiler option.

Note: --vla should not be used together with the longjmp library function, as that can lead to memory leakages.
--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**

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.

**Note:** Any diagnostic messages that have been reclassified as warnings by the option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

This option is not available in the IDE.

See also

`--diag_warning`, page 240.

**Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors**
Descriptions of compiler options
Linker options

This chapter gives detailed reference information about each linker option.

For general syntax rules, see Options syntax, page 227.

Summary of linker options

This table summarizes the linker options:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--basic_heap</td>
<td>Uses a basic heap instead of an advanced heap</td>
</tr>
<tr>
<td>--BE8</td>
<td>Uses the big-endian format BE8</td>
</tr>
<tr>
<td>--BE32</td>
<td>Uses the big-endian format BE32</td>
</tr>
<tr>
<td>--call_graph</td>
<td>Produces a call graph file in XML format</td>
</tr>
<tr>
<td>--config</td>
<td>Specifies the linker configuration file to be used by the linker</td>
</tr>
<tr>
<td>--config_def</td>
<td>Defines symbols for the configuration file</td>
</tr>
<tr>
<td>--cpp_init_routine</td>
<td>Specifies a user-defined C++ dynamic initialization routine</td>
</tr>
<tr>
<td>--cpu</td>
<td>Specifies a processor variant</td>
</tr>
<tr>
<td>--debug_lib</td>
<td>Uses the C-SPY debug library</td>
</tr>
<tr>
<td>--define_symbol</td>
<td>Defines symbols that can be used by the application</td>
</tr>
<tr>
<td>--dependencies</td>
<td>Lists file dependencies</td>
</tr>
<tr>
<td>--diag_error</td>
<td>Treats these message tags as errors</td>
</tr>
<tr>
<td>--diag_remark</td>
<td>Treats these message tags as remarks</td>
</tr>
<tr>
<td>--diag_suppress</td>
<td>Suppresses these diagnostic messages</td>
</tr>
<tr>
<td>--diag_warning</td>
<td>Treats these message tags as warnings</td>
</tr>
<tr>
<td>--diagnostics_tables</td>
<td>Lists all diagnostic messages</td>
</tr>
<tr>
<td>--enable_hardware_workaround</td>
<td>Enables specified hardware workaround</td>
</tr>
<tr>
<td>--enable_stack_usage</td>
<td>Enables stack usage analysis</td>
</tr>
<tr>
<td>--entry</td>
<td>Treats the symbol as a root symbol and as the start of the application</td>
</tr>
</tbody>
</table>

Table 31: Linker options summary
## Summary of linker options

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--error_limit</td>
<td>Specifies the allowed number of errors before linking stops</td>
</tr>
<tr>
<td>--exception_tables</td>
<td>Generates exception tables for C code</td>
</tr>
<tr>
<td>--export_builtin_config</td>
<td>Produces an icf file for the default configuration</td>
</tr>
<tr>
<td>--extra_init</td>
<td>Specifies an extra initialization routine that will be called if it is defined.</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line</td>
</tr>
<tr>
<td>--force_exceptions</td>
<td>Always includes exception runtime code</td>
</tr>
<tr>
<td>--force_output</td>
<td>Produces an output file even if errors occurred</td>
</tr>
<tr>
<td>--image_input</td>
<td>Puts an image file in a section</td>
</tr>
<tr>
<td>--inline</td>
<td>Inlines small routines</td>
</tr>
<tr>
<td>--keep</td>
<td>Forces a symbol to be included in the application</td>
</tr>
<tr>
<td>--log</td>
<td>Enables log output for selected topics</td>
</tr>
<tr>
<td>--log_file</td>
<td>Directs the log to a file</td>
</tr>
<tr>
<td>--mangled_names_in_messages</td>
<td>Adds mangled names in messages</td>
</tr>
<tr>
<td>--map</td>
<td>Produces a map file</td>
</tr>
<tr>
<td>--merge_duplicate_sections</td>
<td>Merges equivalent read-only sections</td>
</tr>
<tr>
<td>--misrac</td>
<td>Enables error messages specific to MISRA-C:1998. This option is a synonym to --misrac1998 and is only available for backwards compatibility.</td>
</tr>
<tr>
<td>--no_dynamic_rtti_elimination</td>
<td>Includes dynamic runtime type information even when it is not needed.</td>
</tr>
<tr>
<td>--no_exceptions</td>
<td>Generates an error if exceptions are used</td>
</tr>
<tr>
<td>--no_fragments</td>
<td>Disables section fragment handling</td>
</tr>
</tbody>
</table>

Table 31: 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 298, --BE32, page 272, and --endian, page 244.

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 298, *--BE8*, page 272, and *--endian*, page 244.

**--call_graph**

Syntax

```
--call_graph {filename|directory}
```

Parameters

For information about specifying a filename or directory, see *Rules for specifying a filename or directory as parameters*, page 228.

Description

Use this option to produce a call graph file. If no filename extension is specified, the extension *cgx* 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

*Stack usage analysis*, page 78

**--config**

Syntax

```
--config filename
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 228.

Description

Use this option to specify the configuration file to be used by the linker (the default filename extension is *icf*). 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 *The linker configuration file*.
Descriptions of linker options

--config_def
Syntax
--config_def symbol[=constant_value]
Parameters
symbol
The name of the symbol to be used in the configuration file.
By default, the value 0 (zero) is used.
constant_value
The constant value of the configuration symbol.
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 define symbol directive in the linker configuration file. This option can be used more than once on the command line.
See also
--define_symbol, page 275 and Interaction between ILINK and the application, page 94.

--cpp_init_routine
Syntax
--cpp_init_routine routine
Parameters
routine
A user-defined C++ dynamic initialization routine.
Description
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.
If any sections with the section type INIT_ARRAY or PREINIT_ARRAY are included in your application, the C++ dynamic initialization routine is considered to be needed. By default, this routine is named __iar_cstart_callctors 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.

To set this option, use Project>Options>Linker>Extra Options.
--cpu

Syntax   --cpu=core

Parameters  

Description  Use this option to select the processor variant to link your application for. The default is to use a processor or architecture compatible with the object file attributes.

See also  --cpu, page 235 for a list of recognized cores and processor macrocells.

Project>Options>General Options>Target>Processor configuration

--define_symbol

Syntax   --define_symbol symbol=constant_value

Parameters  

symbol  The name of the constant symbol that can be used by the application.

constant_value  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 this option is different from the define symbol directive.

See also  --config_def, page 274 and Interaction between ILINK and the application, page 94.

Project>Options>Linker>#define>Defined symbols

--dependencies

Syntax   --dependencies [=i|m] {filename|directory}

Parameters  

i  (default)  Lists only the names of files

m  Lists in makefile style
Descriptions of linker options

For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 228.

**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.

**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:\myproject\foo.o
```

d:\myproject\bar.o

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**
tag
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**
tag
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
--diag_suppress=tag[,tag,...]

Parameters
tag
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>Linker>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax
--diag_warning=tag[,tag,...]

Parameters
tag
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 linker to stop before linking 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
Descriptions of linker options

--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 228.
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
--enable_hardware_workaround=waid[waid[...]]
Parameters
waid
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
--enable_stack_usage
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 --stack_usage_control or --call_graph options, stack usage analysis is automatically enabled.
See also  
*Stack usage analysis, page 78*

Project>Options>Linker>Advanced>Enable stack usage analysis

---**entry**

**Syntax**

`--entry symbol`

**Parameters**

*symbol*  
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

nocreate (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.

unwind  Generates unwind entries that enable an exception to be correctly propagated through functions without exception information.

cantunwind  Generates no-unwind entries so that any attempt to propagate an exception through the function will result in a call to terminate.

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 171.

To set this option, use Project>Options>Linker>Extra Options.

--export_builtin_config

Syntax
--export_builtin_config filename

Parameters
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 228.

Description
Exports the configuration used by default to a file.

This option is not available in the IDE.
--extra_init

Syntax
--extra_init routine

Parameters
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 main is called. Note that the routine must preserve the value passed to it in register R0. No entry is added if the routine is not defined.

To set this option, use Project>Options>Linker>Extra Options.

-f

Syntax
-f filename

Parameters
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 228.

Description
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 171.

---

**--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>Optimizations>C++ Exceptions>Allow>Always include`.

---

**--image_input**

**Syntax**

```
--image_input filename [,symbol,[section[,alignment]]]
```

**Parameters**

- `filename`: The pure binary file containing the raw image you want to link.
- `symbol`: The symbol which the binary data can be referenced with.
- `section`: The section where the binary data will be placed; default is `.text`.
- `alignment`: 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.
Note: Just as for sections from object files, sections created by using the
--image_input option are not included unless actually needed. You can either specify
a symbol in the option and reference this symbol in your application (or by use of a
--keep option), or you can specify a section name and use the keep directive in a linker
configuration file to ensure that the section is included.

Example

--image_input bootstrap.abs,Bootstrap,CSTARTUPCODE,4

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

See also

--keep, page 283.

Project>Options>Linker>Input>Keep symbols
Descriptions of linker options

--log
Syntax
--log topic[,topic,...]
Parameters
topic can be one of:

initialization    Lists copy batches and the compression selected for each batch.
libraries        Lists all decisions taken by the automatic library selector. This might include extra symbols needed (--keep), redirections (--redirect), as well as which runtime libraries that were selected.
modules          Lists each module that is selected for inclusion in the application, and which symbol that caused it to be included.
redirects        Lists redirected symbols.
sections         Lists each symbol and section fragment that is selected for inclusion in the application, and the dependence that caused it to be included.
unused.fragments Lists those section fragments that were not included in the application.
veeners          Lists some veneer creation and usage statistics.

Description     Use this option to make the linker log information to stdout. The log information can be useful for understanding why an executable image became the way it is.
See also         --log_file, page 284.

Project>Options>Linker>List>Generate log

--log_file
Syntax
--log_file filename
Parameters
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 228.

Description     Use this option to direct the log output to the specified file.
--mangled_names_in_messages

Syntax

--mangled_names_in_messages

Description

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, `void h(int, char)` becomes `_Z1hic`.

This option is not available in the IDE.

--map

Syntax

--map `{filename|directory}`

Description

Use this option to produce a linker memory map file. The map file has the default filename extension `map`. The map file contains:

- 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: 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.

The linker considers dynamic runtime type information to be needed if there is a `typeid` or `dynamic_cast` 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.

**Note:** A `typeid` expression for a non-polymorphic type results in a direct reference to a particular RTTI object and will not cause the linker to include any potentially unneeded objects.

See also: Using C++, page 171.

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 171.

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 89.

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*
Descriptions of linker options

--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 DW ARF 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 89.

To set this option, use Project>Options>Linker>Extra Options.
--no_veneers
Syntax --no_veneers
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 Veneers, page 95.
To set this option, use Project>Options>Linker>Extra Options.

--no_vfe
Syntax --no_vfe
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 --vfe, page 294 and Virtual Function Elimination, page 192.
To set related options, choose:
Project>Options>Linker>Advanced>PerformC++ Virtual Function Elimination

--no_warnings
Syntax --no_warnings
Description By default, the linker issues warning messages. Use this option to disable all warning messages.
This option is not available in the IDE.
Descriptions of linker options

---

--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 228.

**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 95.

To set this option, use Project>Options>Linker>Extra Options.

--place_holder

Syntax  

`--place_holder symbol[,size[,section[,alignment]]]`

Parameters  

- `symbol`: The name of the symbol to create
- `size`: Size in ROM; by default 4 bytes
- `section`: Section name to use; by default .text
- `alignment`: 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 ielftool. 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 437.

To set this option, use Project>Options>Linker>Extra Options

--redirect

Syntax  

`--redirect from_symbol=to_symbol`

Parameters  

- `from_symbol`: The name of the source symbol
- `to_symbol`: 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

--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 linker does not generate remarks. Use this option to make the linker generate remarks.

See also

Severity levels, page 225.

Project>Options>Linker>Diagnostics>Enable remarks

--- search

Syntax

--search path

Parameters

path

A path to a directory where the linker should search for object and library files.

Description

Use this option to specify more directories for the linker to search for object and library files in.

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.

See also

The linking process, page 71.

This option is not available in the IDE.

--- semihosting

Syntax

--semihosting [=iar_breakpoint]

Parameters

iar_breakpoint

The IAR-specific mechanism can be used when debugging applications that use SWI/SVC extensively.
**Linker options**

**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 107.

**--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 228.
Descriptions of linker options

**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 78
- 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**

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  
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.

Note: Any diagnostic messages that have been reclassified as warnings by the option --diag_warning will also be treated as errors when --warnings_are_errors is used.

See also  

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 307.

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 234 and `--no_const_align`, page 251.

---

**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 244.

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 300.

**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:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size</th>
<th>Range</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>8 bits</td>
<td>0 to 1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>8 bits</td>
<td>0 to 255</td>
<td>1</td>
</tr>
<tr>
<td>signed char</td>
<td>8 bits</td>
<td>-128 to 127</td>
<td>1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>0 to 255</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 32: Integer types*
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:

```c
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
    DontUseChar=257};
```

For related information, see `--enum_is_int`, page 245.

**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.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size</th>
<th>Range</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed short</td>
<td>16 bits</td>
<td>-32768 to 32767</td>
<td>2</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits</td>
<td>0 to 65535</td>
<td>2</td>
</tr>
<tr>
<td>signed int</td>
<td>32 bits</td>
<td>-2^31 to 2^31-1</td>
<td>4</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32 bits</td>
<td>-2^31 to 2^31-1</td>
<td>4</td>
</tr>
<tr>
<td>signed long</td>
<td>32 bits</td>
<td>-2^31 to 2^31-1</td>
<td>4</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32 bits</td>
<td>0 to 2^32-1</td>
<td>4</td>
</tr>
<tr>
<td>signed long long</td>
<td>64 bits</td>
<td>-2^63 to 2^63-1</td>
<td>8</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64 bits</td>
<td>0 to 2^64-1</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 32: Integer types (Continued)
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 327.
Example

Assume this example:

```c
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.

![Diagram of bitfield allocation](image)

*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:

![Diagram of bitfield_example in little-endian mode]

This is the layout of `bitfield_example` in big-endian mode:

![Diagram of bitfield_example 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range (+/-)</th>
<th>Decimals</th>
<th>Exponent</th>
<th>Manissa</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits</td>
<td>±1.18E-38 to ±3.40E+38</td>
<td>7</td>
<td>8 bits</td>
<td>23 bits</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>±2.23E-308 to ±1.79E+308</td>
<td>15</td>
<td>11 bits</td>
<td>52 bits</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>64 bits</td>
<td>±2.23E-308 to ±1.79E+308</td>
<td>15</td>
<td>11 bits</td>
<td>52 bits</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 33: 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 305.

Floating-point environment

Exception flags for floating-point values are supported for devices with a VFP unit, and they are defined in the \( \text{fenv.h} \) file. For devices without a VFP unit, the functions defined in the \( \text{fenv.h} \) file exist but have no functionality.

The \( \text{feraiseexcept} \) 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:

\[
\begin{array}{cccc}
31 & 30 & 23 & 22 & 0 \\
S & E & M & M & 0 \\
\end{array}
\]

The exponent is 8 bits, and the mantissa is 23 bits.

The value of the number is:

\[
(-1)^S \times 2^{(\text{Exponent}-127)} \times \text{Mantissa}
\]

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.
Part 2. Reference information

Data representation

64-bit floating-point format
The representation of a 64-bit floating-point number as an integer is:

\[
\begin{array}{c|c|c|c}
\text{S} & \text{Exponent} & \text{Mantissa} \\
63 & 62 & 52 & 51 & 0
\end{array}
\]

The exponent is 11 bits, and the mantissa is 52 bits.
The value of the number is:
\[(-1)^S \times 2^{(\text{Exponent}-1023)} \times \text{1.Mantissa}\]
The range of the number is at least:
\[\pm2.23\times10^{-308} \text{ to } \pm1.79\times10^{+308}\]
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 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 subnormal, even though the number is treated as if the exponent was 1. Unlike normal numbers, subnormal numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a subnormal number is:
\[(-1)^S \times 2^{(1-\text{BIAS})} \times 0.\text{Mantissa}\]
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 0x0–0xFFFFF000.
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 a signed 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**

```c
struct First
{
  char c;
  short s;
} s;
```

This diagram shows the layout in memory:

![Figure 18: Structure layout](image)

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.
Structure types

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:

```c
#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](image)

This example declares a new non-packed structure, S2, that contains the structure s declared in the previous example:

```c
struct S2
{
  struct S s;
  long l;
};
```

S2 has this memory layout:

![Figure 20: Packed structure layout](image)
The structure $S$ will use the memory layout, size, and alignment described in the previous example. The alignment of the member $l$ is 4, which means that alignment of the structure $S_2$ will become 4.

For more information, see Alignment of elements in a structure, page 199.

### 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:

  ```c
  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:

```c
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.

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 242 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
General syntax rules for extended keywords

- **Data type attributes**: __big_endian, __little_endian, and __packed.

You can specify as many type attributes as required for each level of pointer indirection.

**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:

```c
__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:

```c
#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`:

```c
int __packed * p;         // A pointer to a packed integer.
int * __packed p;         // A packed pointer to an integer.
__packed int * p;         // 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:

```c
__irq __arm void my_handler(void);
```

or

```c
void (__irq __arm my_handler)(void);
```
This declaration of \texttt{my\_handler} is equivalent with the previous one:

\begin{verbatim}
#pragma type\_attribute=\_irq \_arm
void my\_handler(void);
\end{verbatim}

\section*{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:

\begin{itemize}
  \item Object attributes that can be used for variables: \_\_absolute, \_\_no\_init
  \item Object attributes that can be used for functions and variables: location, \&, \_\_root, and \_\_weak
  \item Object attributes that can be used for functions: \_\_intrinsic, \_\_nested, \_\_noreturn, \_\_ramfunc, and \_\_stackless.
\end{itemize}

You can specify as many object attributes as required for a specific function or data object.

For more information about \texttt{location} and \&, see \textit{Controlling data and function placement in memory}, page 201.

\section*{Syntax for object attributes}

The object attribute must be placed in front of the type. For example, to place \texttt{myarray} in memory that is not initialized at startup:

\begin{verbatim}
\_\_no\_init int myarray[10];
\end{verbatim}

The \texttt{#pragma object\_attribute} directive can also be used. This declaration is equivalent to the previous one:

\begin{verbatim}
#pragma object\_attribute=\_\_no\_init
int myarray[10];
\end{verbatim}

\textbf{Note}: Object attributes cannot be used in combination with the \texttt{typedef} keyword.

\section*{Summary of extended keywords}

This table summarizes the extended keywords:

\begin{center}
\begin{tabular}{|l|p{12cm}|}
\hline
Extended keyword & Description \\
\hline
\_\_absolute & Makes references to the object use absolute addressing \\
\hline
\end{tabular}
\end{center}

\textit{Table 34: Extended keywords summary}
### Extended keywords

<table>
<thead>
<tr>
<th>Extended keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__arm</td>
<td>Makes a function execute in ARM mode</td>
</tr>
<tr>
<td>__big_endian</td>
<td>Declares a variable to use the big-endian byte order</td>
</tr>
<tr>
<td>__fiq</td>
<td>Declares a fast interrupt function</td>
</tr>
<tr>
<td>__interwork</td>
<td>Declares a function to be callable from both ARM and Thumb mode</td>
</tr>
<tr>
<td>__intrinsic</td>
<td>Reserved for compiler internal use only</td>
</tr>
<tr>
<td>__irq</td>
<td>Declares an interrupt function</td>
</tr>
<tr>
<td>__little_endian</td>
<td>Declares a variable to use the little-endian byte order</td>
</tr>
<tr>
<td>__nested</td>
<td>Allows an __irq declared interrupt function to be nested, that is, interruptible by the same type of interrupt</td>
</tr>
<tr>
<td>__no_init</td>
<td>Places a data object in non-volatile memory</td>
</tr>
<tr>
<td>__noreturn</td>
<td>Informs the compiler that the function will not return</td>
</tr>
<tr>
<td>__packed</td>
<td>Decreases data type alignment to 1</td>
</tr>
<tr>
<td>__pcrel</td>
<td>Used internally by the compiler for constant data when the --ropi compiler option is used</td>
</tr>
<tr>
<td>__ramfunc</td>
<td>Makes a function execute in RAM</td>
</tr>
<tr>
<td>__root</td>
<td>Ensures that a function or variable is included in the object code even if unused</td>
</tr>
<tr>
<td>__sbrel</td>
<td>Used internally by the compiler for constant data when the --rwpi compiler option is used</td>
</tr>
<tr>
<td>__stackless</td>
<td>Makes a function callable without a working stack</td>
</tr>
<tr>
<td>__swi</td>
<td>Declares a software interrupt function</td>
</tr>
<tr>
<td>__task</td>
<td>Relaxes the rules for preserving registers</td>
</tr>
<tr>
<td>__thumb</td>
<td>Makes a function execute in Thumb mode</td>
</tr>
<tr>
<td>__weak</td>
<td>Declares a symbol to be externally weakly linked</td>
</tr>
</tbody>
</table>

Table 34: Extended keywords summary (Continued)

---

**Descriptions of extended keywords**

These sections give detailed information about each extended keyword.

**__absolute**

Syntax: 

Follows the generic syntax rules for object attributes, see *Object attributes*, page 315.
Description
The `__absolute` keyword makes references to the object use absolute addressing.
The following limitations apply:
- Only available when the `--ropi` or `--rwpi` compiler option is used
- Can only be used on external declarations.

Example
```c
extern __absolute char otherBuffer[100];
```

__arm

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 313.

Description
The `__arm` keyword makes a function execute in ARM mode. An `__arm` declared function can, unless it is also declared `__interwork`, only be called from functions that also execute in ARM mode.

A function declared `__arm` cannot be declared `__thumb`.

Note: Non-interwork ARM functions cannot be called from Thumb mode.

Example
```c
__arm int func1(void);
```

See also
`__interwork`, page 318.

__big_endian

Syntax
Follows the generic syntax rules for type attributes that can be used on data objects, see Type attributes, page 313.

Description
The `__big_endian` 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 `__big_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
```c
__big_endian long my_variable;
```

See also
`__little_endian`, page 319.
Descriptions of extended keywords

__fiq
Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see
_Type attributes_, page 313.

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 313.

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 313.

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);
See also  

__little_endian

Syntax  

Follows the generic syntax rules for type attributes that can be used on data objects, see Type attributes, page 313.

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 317.

__nested

Syntax  

Follows the generic syntax rules for object attributes, see Object attributes, page 315.

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 61 and --align_sp_on_irq, page 234.

__no_init

Syntax  

Follows the generic syntax rules for object attributes, see Object attributes, page 315.

Description  

Use the __no_init 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.
Descriptions of extended keywords

Example

```c
__no_init int myarray[10];
```

See also

*Do not initialize directive*, page 410.

__noreturn__

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 315.

Description

The __noreturn 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 `abort` and `exit`.

Example

```c
__noreturn void terminate(void);
```

__packed__

Syntax

Follows the generic syntax rules for type attributes that can be used on data, see *Type attributes*, page 313.

Description

Use the __packed keyword to decrease the data type alignment to 1. __packed can be used for two purposes:

- When used with a `struct` or `union` type definition, the maximum alignment of members of that `struct` or `union` is set to 1, to eliminate any gaps between the members. The type of each members also receives the __packed type attribute.
- When used with any other type, the resulting type is the same as the type without the __packed type attribute, but with an alignment of 1. Types that already have an alignment of 1 are not affected by the __packed type attribute.

A normal pointer can be implicitly converted to a pointer to __packed, but the reverse conversion requires a cast.

**Note:** Accessing data types at other alignments than their natural alignment can result in code that is significantly larger and slower.

Use either __packed or #pragma pack to relax the alignment restrictions for a type and the objects defined using that type. Mixing __packed and #pragma pack might lead to unexpected behavior.
Extended keywords

__packed

Example

```c
__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 336.
```

__ramfunc

Syntax

Follows the generic syntax rules for object attributes, see Object attributes, page 315.

Description

The __ramfunc keyword makes a function execute in RAM. Two code sections will be created: one for the RAM execution (.text.rw), 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 .text.rw.

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 315.

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 89.
__stackless

Syntax
Follows the generic syntax rules for object attributes, see Object attributes, page 315.

Description
The __stackless keyword declares a function that can be called without a working stack.

A function declared __stackless 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.

Example
__stackless void start_application(void);

__swi

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 313.

Description
The __swi declares a software interrupt function. It inserts an SVC (formerly SWI) instruction and the specified software interrupt number to make a proper function call.

A function declared __swi accepts arguments and returns values. The __swi 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.

The __swi keyword also expects a software interrupt number which is specified with the #pragma swi_number=number directive. The swi_number is used as an argument to the generated assembler SVC instruction, and can be used by the SVC interrupt handler, for example SWI_Handler, 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.

Note: All interrupt functions must be compiled in ARM mode, except for Cortex-M. Use either the __arm keyword or the #pragma type_attribute=__arm directive to alter the default behavior if needed.

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:

```c
...  
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:

```c
...  
__swi __arm int the_actual_swi0x23_function(int a, int b)  
{
  ...  
  return 42;
}
```

See also

*Software interrupts*, page 62 and *Calling convention*, page 150.

**__task**

**Syntax**

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 313.

**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);
```
Descriptions of extended keywords

**__thumb**

**Syntax**

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 313.

**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**

```c
__thumb int func2(void);
```

**See also**

__interwork, page 318.

**__weak**

**Syntax**

Follows the generic syntax rules for object attributes, see *Object attributes*, page 315.

**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**

```c
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:

<table>
<thead>
<tr>
<th>Pragma directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitfields</td>
<td>Controls the order of bitfield members</td>
</tr>
<tr>
<td>calls</td>
<td>Lists possible called functions for indirect calls</td>
</tr>
<tr>
<td>call_graph_root</td>
<td>Specifies that the function is a call graph root</td>
</tr>
<tr>
<td>data_alignment</td>
<td>Gives a variable a higher (more strict) alignment</td>
</tr>
<tr>
<td>diag_default</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_error</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_remark</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_suppress</td>
<td>Suppresses diagnostic messages</td>
</tr>
<tr>
<td>diag_warning</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>error</td>
<td>Signals an error while parsing</td>
</tr>
<tr>
<td>include_alias</td>
<td>Specifies an alias for an include file</td>
</tr>
<tr>
<td>inline</td>
<td>Controls inlining of a function</td>
</tr>
<tr>
<td>language</td>
<td>Controls the IAR Systems language extensions</td>
</tr>
</tbody>
</table>

Table 35: Pragma directives summary
Summary of pragma directives

<table>
<thead>
<tr>
<th>Pragma directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Specifies the absolute address of a variable, places a variable in a register, or places groups of functions or variables in named sections</td>
</tr>
<tr>
<td>message</td>
<td>Prints a message</td>
</tr>
<tr>
<td>object_attribute</td>
<td>Changes the definition of a variable or a function</td>
</tr>
<tr>
<td>optimize</td>
<td>Specifies the type and level of an optimization</td>
</tr>
<tr>
<td>pack</td>
<td>Specifies the alignment of structures and union members</td>
</tr>
<tr>
<td>__printf_args</td>
<td>Verifies that a function with a printf-style format string is called with the correct arguments</td>
</tr>
<tr>
<td>required</td>
<td>Ensures that a symbol that is needed by another symbol is included in the linked output</td>
</tr>
<tr>
<td>rtmodel</td>
<td>Adds a runtime model attribute to the module</td>
</tr>
<tr>
<td>__scanf_args</td>
<td>Verifies that a function with a scanf-style format string is called with the correct arguments</td>
</tr>
<tr>
<td>section</td>
<td>Declares a section name to be used by intrinsic functions</td>
</tr>
<tr>
<td>segment</td>
<td>This directive is an alias for #pragma section</td>
</tr>
<tr>
<td>STDC CX_LIMITED_RANGE</td>
<td>Specifies whether the compiler can use normal complex mathematical formulas or not</td>
</tr>
<tr>
<td>STDC FENV_ACCESS</td>
<td>Specifies whether your source code accesses the floating-point environment or not</td>
</tr>
<tr>
<td>STDC FP_CONTRACT</td>
<td>Specifies whether the compiler is allowed to contract floating-point expressions or not</td>
</tr>
<tr>
<td>swi_number</td>
<td>Sets the interrupt number of a software interrupt function</td>
</tr>
<tr>
<td>type_attribute</td>
<td>Changes the declaration and definitions of a variable or function</td>
</tr>
<tr>
<td>weak</td>
<td>Makes a definition a weak definition, or creates a weak alias for a function or a variable</td>
</tr>
</tbody>
</table>

Table 35: Pragma directives summary (Continued)

**Note**: For portability reasons, the pragma directives `alignment`, `baseaddr`, `codeseg`, `constseg`, `dataneg`, `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 474.
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 *Bitfields*, page 300.

- **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 `reversed_disjoint_types`.

- **default**
  Restores to default layout of bitfield members. The default behavior for the compiler is `joined_types`.

**Description**

Use this pragma directive to control the layout of bitfield members.

**Example**

```c
#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. */
```

**See also**

*Bitfields*, page 300.
### calls

**Syntax**

```
#pragma calls=function[, function...]
```

**Parameters**

- `function` 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**

```c
void Fun1(), Fun2();
void Caller(void (*fp)(void))
{
    #pragma calls = Fun1, Fun2
    (*fp)();
}
```

**See also**

*Stack usage analysis, page 78*

### call_graph_root

**Syntax**

```
#pragma call_graph_root=[category]
```

**Parameters**

- `category` 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 `#pragma call_graph_root` directive on such a function you will override the default category. You can specify any string as a category.

**Example**

```
#pragma call_graph_root="interrupt"
```

**See also**

*Stack usage analysis, page 78*
data_alignment

Syntax

#pragma data_alignment=expression

Parameters

expression A constant which must be a power of two (1, 2, 4, etc.).

Description

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.

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.

Note: Normally, the size of a variable is a multiple of its alignment. The data_alignment 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.

diag_default

Syntax

#pragma diag_default=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 back to the default, or to the severity level defined on the command line by any of the options --diag_error, --diagRemark, --diag_suppress, or --diag_warnings, for the diagnostic messages specified with the tags.

See also

Diagnostics, page 224.

diag_error

Syntax

#pragma diag_error=tag[,tag,...]

Parameters

tag The number of a diagnostic message, for example the message number Pe177.
<table>
<thead>
<tr>
<th>Description</th>
<th>Use this pragma directive to change the severity level to <strong>error</strong> for the specified diagnostics.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>See also</strong></td>
<td><em>Diagnostics</em>, page 224.</td>
</tr>
</tbody>
</table>

### diag_remark

**Syntax**  
```c
#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 **remark** for the specified diagnostic messages.

**See also**  
*Diagnostics*, page 224.

### diag_suppress

**Syntax**  
```c
#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**  
*Diagnostics*, page 224.

### diag_warning

**Syntax**  
```c
#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 **warning** for the specified diagnostic messages.
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>
### inline

**Syntax**

```c
#pragma inline [=forced|=never]
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No parameter</td>
<td>Has the same effect as the <code>inline</code> keyword.</td>
</tr>
<tr>
<td>forced</td>
<td>Disables the compiler's heuristics and forces inlining.</td>
</tr>
<tr>
<td>never</td>
<td>Disables the compiler's heuristics and makes sure that the function will not be inlined.</td>
</tr>
</tbody>
</table>

**Description**

Use `#pragma inline` to advise the compiler that the function defined immediately after the directive should be inlined according to C++ inline semantics.

Specifying `#pragma inline=forced` will always inline the defined function. If the compiler fails to inline the 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 also on the Medium optimization level.

**See also**

`Inlining functions`, page 65

### language

**Syntax**

```c
#pragma language={extended|default|save|restore}
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extended</td>
<td>Enables the IAR Systems language extensions from the first use of the pragma directive and onward.</td>
</tr>
<tr>
<td>default</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 221.
Pragma directives

**Description**
Use this pragma directive to control the use of language extensions.

**Example**
At the top of a file that needs to be compiled with IAR Systems extensions enabled:

```
#pragma language=extended
/* The rest of the file. */
```

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:

```
#pragma language=save
#pragma language=extended
/* Part of source code. */
#pragma language=restore
```

**See also**
-e, page 242 and --strict, page 265.

### location

**Syntax**
```
#pragma location={address|register|NAME}
```

**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.
- `NAME`: A user-defined section name; cannot be a section name predefined for use by the compiler and linker.

**Description**
Use this pragma directive to specify:

- The location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variables must be declared `__no_init`.

save|restore

Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code.

Each use of `save` must be followed by a matching `restore` in the same file without any intervening `#include` directive.
Descriptions of pragma directives

- An identifier specifying a register. The variable defined after the pragma directive is placed in the register. The variable must be declared as __no_init and have file scope.

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 __no_init and variables declared as const) in the same named section.

Example

```
#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 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 */
```

See also

*Controlling data and function placement in memory*, page 201.

message

Syntax

```
#pragma message(message)
```

Parameters

- `message` 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 315.

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 313.

optimize

Syntax
#pragma optimize=[goal][level][no_optimization...]

Parameters

\( goal \)
Choose between:
size, optimizes for size
balanced, optimizes balanced between speed and size
speed, optimizes for speed.
no_size_constraints, optimizes for speed, but relaxes the normal restrictions for code size expansion.

\( level \)
Specifies the level of optimization; choose between none, low, medium, or high.
no_optimization  Disables one or several optimizations; choose between:
  no_code_motion, disables code motion
  no_cse, disables common subexpression elimination
  no_inline, disables function inlining
  no_tbaa, disables type-based alias analysis
  no_unroll, disables loop unrolling
  no_scheduling, disables instruction scheduling.

Description  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.

The parameters size, balanced, speed, and no_size_constraints only have effect on the high 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.

Note: If you use the #pragma optimize directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```c
#pragma optimize=speed
int SmallAndUsedOften()
{
  /* Do something here. */
}
```

```c
#pragma optimize=size
int BigAndSeldomUsed()
{
  /* Do something here. */
}
```

See also  Fine-tuning enabled transformations, page 208.
**Pragma directives**

### pack

**Syntax**

```c
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop},[name] [,n])
```

**Parameters**

- `n` Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16
- Empty list Restores the structure alignment to default
- `push` Sets a temporary structure alignment
- `pop` Restores the structure alignment from a temporarily pushed alignment
- `name` An optional pushed or popped alignment label

**Description**

Use this pragma directive to specify the maximum alignment of `struct` and `union` members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or the end of the compilation unit.

**Note:** This can result in significantly larger and slower code when accessing members of the structure.

Use either `__packed` or `#pragma pack` to relax the alignment restrictions for a type and the objects defined using that type. Mixing `__packed` and `#pragma pack` might lead to unexpected behavior.

**See also**


### __printf_args

**Syntax**

```c
#pragma __printf_args
```

**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 `%d`) is syntactically correct.
Descriptions of pragma directives

Example

```c
#pragma __printf_args
int printf(char const *,...);

void PrintNumbers(unsigned short x)
{
    printf("%d", x); /* Compiler checks that x is an integer */
}
```

**required**

**Syntax**

```c
#pragma required=symbol
```

**Parameters**

- `symbol` Any statically linked function or variable.

**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**

```c
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**

```c
#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.
Part 2. Reference information

Pragma directives

**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 136.

---

**__scanf_args**

**Syntax**

```
#pragma __scanf_args
```

**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**

```
#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;
}
```
Descriptions of pragma directives

section

Syntax

    #pragma section="NAME"

alias

    #pragma segment="NAME"

Parameters

    NAME                The name of the section.

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

    #pragma section="MYSECTION"

See also

    Dedicated section operators, page 165. For more information about sections and
    segment parts, see the chapter Linking your application.

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.
Pragmas directives

**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**

- **number**: The software interrupt number
## Description of Pragma Directives

### swi_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 62

### 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 313.

**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 symbol1={symbol2}
```

**Parameters**
- `symbol1` A function or variable with external linkage.

**See also**
IAR C/C++ Development Guide 342 Compiling and Linking for ARM
Pragma directives

Description
This pragma directive can be used in one of two ways:

- To make the definition of a function or variable with external linkage a weak definition. The `__weak` 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
To make the definition of `foo` a weak definition, write:

```c
#pragma weak foo
```

To make `NMI_Handler` a weak alias for `Default_Handler`, write:

```c
#pragma weak NMI_Handler=Default_Handler
```

If `NMI_Handler` is not defined elsewhere in the program, all references to `NMI_Handler` will refer to `Default_Handler`.

See also `__weak`, page 324.
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 351.

Note that the intrinsic function names start with double underscores, for example:

```c
__disable_interrupt
```

This table summarizes the intrinsic functions:

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__CLREX</td>
<td>Inserts a CLREX instruction</td>
</tr>
<tr>
<td>__CLZ</td>
<td>Inserts a CLZ instruction</td>
</tr>
<tr>
<td>__disable_fiq</td>
<td>Disables fast interrupt requests (fiq)</td>
</tr>
<tr>
<td>__disable_interrupt</td>
<td>Disables interrupts</td>
</tr>
<tr>
<td>__disable_irq</td>
<td>Disables interrupt requests (irq)</td>
</tr>
<tr>
<td>__DMB</td>
<td>Inserts a DMB instruction</td>
</tr>
<tr>
<td>__DSB</td>
<td>Inserts a DSB instruction</td>
</tr>
<tr>
<td>__enable_fiq</td>
<td>Enables fast interrupt requests (fiq)</td>
</tr>
<tr>
<td>__enable_interrupt</td>
<td>Enables interrupts</td>
</tr>
<tr>
<td>__enable_irq</td>
<td>Enables interrupt requests (irq)</td>
</tr>
<tr>
<td>__get_BASEPRI</td>
<td>Returns the value of the Cortex-M3/Cortex-M4 BASEPRI register</td>
</tr>
<tr>
<td>__get_CONTROL</td>
<td>Returns the value of the Cortex-M CONTROL register</td>
</tr>
</tbody>
</table>

Table 36: Intrinsic functions summary
### Summary of intrinsic functions

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__get_CPSR</td>
<td>Returns the value of the ARM CPSR (Current Program Status Register)</td>
</tr>
<tr>
<td>__get_FAULTMASK</td>
<td>Returns the value of the Cortex-M3/Cortex-M4 FAULTMASK register</td>
</tr>
<tr>
<td>__get_FPSCR</td>
<td>Returns the value of FPSCR</td>
</tr>
<tr>
<td>__get_interrupt_state</td>
<td>Returns the interrupt state</td>
</tr>
<tr>
<td>__get_IPSR</td>
<td>Returns the value of the IPSR register</td>
</tr>
<tr>
<td>__get_LR</td>
<td>Returns the value of the link register</td>
</tr>
<tr>
<td>__get_MSP</td>
<td>Returns the value of the MSP register</td>
</tr>
<tr>
<td>__get_PRIMASK</td>
<td>Returns the value of the Cortex-M PRIMASK register</td>
</tr>
<tr>
<td>__get_PSP</td>
<td>Returns the value of the PSP register</td>
</tr>
<tr>
<td>__get_PSR</td>
<td>Returns the value of the PSR register</td>
</tr>
<tr>
<td>__get_SB</td>
<td>Returns the value of the static base register</td>
</tr>
<tr>
<td>__get_SP</td>
<td>Returns the value of the stack pointer register</td>
</tr>
<tr>
<td>__ISB</td>
<td>Inserts an ISB instruction</td>
</tr>
<tr>
<td>__LDC</td>
<td>Inserts the coprocessor load instruction LDC</td>
</tr>
<tr>
<td>__LDCL</td>
<td>Inserts the coprocessor load instruction LDCL</td>
</tr>
<tr>
<td>__LDC2</td>
<td>Inserts the coprocessor load instruction LDC2</td>
</tr>
<tr>
<td>__LDC2L</td>
<td>Inserts the coprocessor load instruction LDC2L</td>
</tr>
<tr>
<td>__LDC_nidx</td>
<td>Inserts the coprocessor load instruction LDC</td>
</tr>
<tr>
<td>__LDCL_nidx</td>
<td>Inserts the coprocessor load instruction LDCL</td>
</tr>
<tr>
<td>__LDC2_nidx</td>
<td>Inserts the coprocessor load instruction LDC2</td>
</tr>
<tr>
<td>__LDC2L_nidx</td>
<td>Inserts the coprocessor load instruction LDC2L</td>
</tr>
<tr>
<td>__LDREX</td>
<td>Inserts an LDREX instruction</td>
</tr>
<tr>
<td>__LDREXB</td>
<td>Inserts an LDREXB instruction</td>
</tr>
<tr>
<td>__LDREXD</td>
<td>Inserts an LDREXD instruction</td>
</tr>
<tr>
<td>__LDREXH</td>
<td>Inserts an LDREXH instruction</td>
</tr>
<tr>
<td>__MCR</td>
<td>Inserts the coprocessor write instruction MCR</td>
</tr>
<tr>
<td>__MCR2</td>
<td>Inserts the coprocessor write instruction MCR2</td>
</tr>
<tr>
<td>__MRC</td>
<td>Inserts the coprocessor read instruction MRC</td>
</tr>
<tr>
<td>__MRC2</td>
<td>Inserts the coprocessor read instruction MRC2</td>
</tr>
<tr>
<td>__no_operation</td>
<td>Inserts a NOP instruction</td>
</tr>
</tbody>
</table>

*Table 36: Intrinsic functions summary (Continued)*
<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__PKHTB</td>
<td>Inserts a PKHTB instruction</td>
</tr>
<tr>
<td>__PKHTB</td>
<td>Inserts a PKHTB instruction</td>
</tr>
<tr>
<td>__PLD</td>
<td>Inserts the preload data instruction PLD</td>
</tr>
<tr>
<td>__PLDW</td>
<td>Inserts the preload data instruction PLDW</td>
</tr>
<tr>
<td>__PLI</td>
<td>Inserts a PLI instruction</td>
</tr>
<tr>
<td>__QADD</td>
<td>Inserts a QADD instruction</td>
</tr>
<tr>
<td>__QADD8</td>
<td>Inserts a QADD8 instruction</td>
</tr>
<tr>
<td>__QADD16</td>
<td>Inserts a QADD16 instruction</td>
</tr>
<tr>
<td>__QASX</td>
<td>Inserts a QASX instruction</td>
</tr>
<tr>
<td>__QFlag</td>
<td>Returns the value of the cumulative saturation flag of the FPSCR register</td>
</tr>
<tr>
<td>__QDADD</td>
<td>Inserts a QDADD instruction</td>
</tr>
<tr>
<td>__QDOUBLE</td>
<td>Inserts a QDOUBLE instruction</td>
</tr>
<tr>
<td>__QDSUB</td>
<td>Inserts a QDSUB instruction</td>
</tr>
<tr>
<td>__QFlag</td>
<td>Returns the Q flag that indicates if overflow/saturation has occurred</td>
</tr>
<tr>
<td>__QSAX</td>
<td>Inserts a QSAX instruction</td>
</tr>
<tr>
<td>__SADD8</td>
<td>Inserts an SADD8 instruction</td>
</tr>
<tr>
<td>__SADD16</td>
<td>Inserts an SADD16 instruction</td>
</tr>
<tr>
<td>__QASX</td>
<td>Inserts a QASX instruction</td>
</tr>
<tr>
<td>__SASX</td>
<td>Inserts an SASX instruction</td>
</tr>
<tr>
<td>__SEL</td>
<td>Inserts an SEL instruction</td>
</tr>
<tr>
<td>__QFlag</td>
<td>Returns the Q flag that indicates if overflow/saturation has occurred</td>
</tr>
<tr>
<td>__QFlag</td>
<td>Clears the value of the cumulative saturation flag QC of the FPSCR register</td>
</tr>
<tr>
<td>__REV</td>
<td>Inserts an REV instruction</td>
</tr>
<tr>
<td>__REV16</td>
<td>Inserts an REV16 instruction</td>
</tr>
<tr>
<td>__REVSH</td>
<td>Inserts an REVSH instruction</td>
</tr>
<tr>
<td>__SADD8</td>
<td>Inserts an SADD8 instruction</td>
</tr>
<tr>
<td>__SADD16</td>
<td>Inserts an SADD16 instruction</td>
</tr>
<tr>
<td>__SASX</td>
<td>Inserts an SASX instruction</td>
</tr>
<tr>
<td>__SEL</td>
<td>Inserts an SEL instruction</td>
</tr>
</tbody>
</table>

Table 36: Intrinsic functions summary (Continued)
### Summary of intrinsic functions

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__set_BASEPRI</td>
<td>Sets the value of the Cortex-M3/Cortex-M4 BASEPRI register</td>
</tr>
<tr>
<td>__set_CONTROL</td>
<td>Sets the value of the Cortex-M CONTROL register</td>
</tr>
<tr>
<td>__set_CPSR</td>
<td>Sets the value of the ARM CPSR (Current Program Status Register)</td>
</tr>
<tr>
<td>__set_FAULTMASK</td>
<td>Sets the value of the Cortex-M3/Cortex-M4 FAULTMASK register</td>
</tr>
<tr>
<td>__set_FPSCR</td>
<td>Sets the value of the FPSCR register</td>
</tr>
<tr>
<td>__set_interrupt_state</td>
<td>Restores the interrupt state</td>
</tr>
<tr>
<td>__set_LR</td>
<td>Assigns a new address to the link register</td>
</tr>
<tr>
<td>__set_MSP</td>
<td>Sets the value of the MSP register</td>
</tr>
<tr>
<td>__set_PRIMASK</td>
<td>Sets the value of the Cortex-M PRIMASK register</td>
</tr>
<tr>
<td>__set_PSP</td>
<td>Sets the value of the PSP register</td>
</tr>
<tr>
<td>__set_SB</td>
<td>Assigns a new address to the static base register</td>
</tr>
<tr>
<td>__set_SP</td>
<td>Assigns a new address to the stack pointer register</td>
</tr>
<tr>
<td>__SEV</td>
<td>Inserts an SEV instruction</td>
</tr>
<tr>
<td>__SHADD8</td>
<td>Inserts an SHADD8 instruction</td>
</tr>
<tr>
<td>__SHADD16</td>
<td>Inserts an SHADD16 instruction</td>
</tr>
<tr>
<td>__SHASX</td>
<td>Inserts an SHASX instruction</td>
</tr>
<tr>
<td>__SHSAX</td>
<td>Inserts an SHSAX instruction</td>
</tr>
<tr>
<td>__SHSUB8</td>
<td>Inserts an SHSUB8 instruction</td>
</tr>
<tr>
<td>__SHSUB16</td>
<td>Inserts an SHSUB16 instruction</td>
</tr>
<tr>
<td>__SMLABB</td>
<td>Inserts an SMLABB instruction</td>
</tr>
<tr>
<td>__SMLABT</td>
<td>Inserts an SMLABT instruction</td>
</tr>
<tr>
<td>__SMLAD</td>
<td>Inserts an SMLAD instruction</td>
</tr>
<tr>
<td>__SMLADX</td>
<td>Inserts an SMLADX instruction</td>
</tr>
<tr>
<td>__SMLALBB</td>
<td>Inserts an SMLALBB instruction</td>
</tr>
<tr>
<td>__SMLALBT</td>
<td>Inserts an SMLALBT instruction</td>
</tr>
<tr>
<td>__SMLALD</td>
<td>Inserts an SMLALD instruction</td>
</tr>
<tr>
<td>__SMLALDX</td>
<td>Inserts an SMLALDX instruction</td>
</tr>
<tr>
<td>__SMLALTB</td>
<td>Inserts an SMLALTB instruction</td>
</tr>
<tr>
<td>__SMLALT</td>
<td>Inserts an SMLALT instruction</td>
</tr>
<tr>
<td>__SMLATB</td>
<td>Inserts an SMLATB instruction</td>
</tr>
</tbody>
</table>

*Table 36: Intrinsic functions summary (Continued)*
<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__SMLATT</td>
<td>Inserts an SMLATT instruction</td>
</tr>
<tr>
<td>__SMLAWB</td>
<td>Inserts an SMLAWB instruction</td>
</tr>
<tr>
<td>__SMLAWT</td>
<td>Inserts an SMLAWT instruction</td>
</tr>
<tr>
<td>__SMLSD</td>
<td>Inserts an SMLSD instruction</td>
</tr>
<tr>
<td>__SMLSDX</td>
<td>Inserts an SMLSDX instruction</td>
</tr>
<tr>
<td>__SMLSLD</td>
<td>Inserts an SMLSLD instruction</td>
</tr>
<tr>
<td>__SMLSLDX</td>
<td>Inserts an SMLSLDX instruction</td>
</tr>
<tr>
<td>__SMMLA</td>
<td>Inserts an SMMLA instruction</td>
</tr>
<tr>
<td>__SMMLAR</td>
<td>Inserts an SMMLAR instruction</td>
</tr>
<tr>
<td>__SMMLS</td>
<td>Inserts an SMMLS instruction</td>
</tr>
<tr>
<td>__SMMLSR</td>
<td>Inserts an SMMLSR instruction</td>
</tr>
<tr>
<td>__SMMUL</td>
<td>Inserts an SMMUL instruction</td>
</tr>
<tr>
<td>__SMMULR</td>
<td>Inserts an SMMULR instruction</td>
</tr>
<tr>
<td>__SMUAD</td>
<td>Inserts an SMUAD instruction</td>
</tr>
<tr>
<td>__SMUADX</td>
<td>Inserts an SMUADX instruction</td>
</tr>
<tr>
<td>__SMUL</td>
<td>Inserts a signed 16-bit multiplication</td>
</tr>
<tr>
<td>__SMULBB</td>
<td>Inserts an SMULBB instruction</td>
</tr>
<tr>
<td>__SMULBT</td>
<td>Inserts an SMULBT instruction</td>
</tr>
<tr>
<td>__SMULTB</td>
<td>Inserts an SMULTB instruction</td>
</tr>
<tr>
<td>__SMULTT</td>
<td>Inserts an SMULTT instruction</td>
</tr>
<tr>
<td>__SMULNB</td>
<td>Inserts an SMULNB instruction</td>
</tr>
<tr>
<td>__SMULNT</td>
<td>Inserts an SMULNT instruction</td>
</tr>
<tr>
<td>__SMUSD</td>
<td>Inserts an SMUSD instruction</td>
</tr>
<tr>
<td>__SMUSDX</td>
<td>Inserts an SMUSDX instruction</td>
</tr>
<tr>
<td>__SSAT</td>
<td>Inserts an SSAT instruction</td>
</tr>
<tr>
<td>__SSAT16</td>
<td>Inserts an SSAT16 instruction</td>
</tr>
<tr>
<td>__SSAX</td>
<td>Inserts an SSAX instruction</td>
</tr>
<tr>
<td>__SSUB8</td>
<td>Inserts an SSUB8 instruction</td>
</tr>
<tr>
<td>__SSUB16</td>
<td>Inserts an SSUB16 instruction</td>
</tr>
<tr>
<td>__STC</td>
<td>Inserts the coprocessor store instruction STC</td>
</tr>
<tr>
<td>__STCL</td>
<td>Inserts the coprocessor store instruction STCL</td>
</tr>
<tr>
<td>__STC2</td>
<td>Inserts the coprocessor store instruction STC2</td>
</tr>
</tbody>
</table>

*Table 36: Intrinsic functions summary (Continued)*
## Summary of intrinsic functions

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__STC2L</td>
<td>Inserts the coprocessor store instruction STC2L</td>
</tr>
<tr>
<td>__STC_noidx</td>
<td>Inserts the coprocessor store instruction STC</td>
</tr>
<tr>
<td>__STCL_noidx</td>
<td>Inserts the coprocessor store instruction STCL</td>
</tr>
<tr>
<td>__STC2_noidx</td>
<td>Inserts the coprocessor store instruction STC2</td>
</tr>
<tr>
<td>__STC2L_noidx</td>
<td>Inserts the coprocessor store instruction STC2L</td>
</tr>
<tr>
<td>__STREX</td>
<td>Inserts a STREX instruction</td>
</tr>
<tr>
<td>__STREXB</td>
<td>Inserts a STREXB instruction</td>
</tr>
<tr>
<td>__STREXD</td>
<td>Inserts a STREXD instruction</td>
</tr>
<tr>
<td>__STREXH</td>
<td>Inserts a STREXH instruction</td>
</tr>
<tr>
<td>__SWP</td>
<td>Inserts an SNP instruction</td>
</tr>
<tr>
<td>__SWPB</td>
<td>Inserts an SNPB instruction</td>
</tr>
<tr>
<td>__SXTAB</td>
<td>Inserts an SXTAB instruction</td>
</tr>
<tr>
<td>__SXTAB16</td>
<td>Inserts an SXTAB16 instruction</td>
</tr>
<tr>
<td>__SXTAH</td>
<td>Inserts an SXTAH instruction</td>
</tr>
<tr>
<td>__SXTB16</td>
<td>Inserts an SXTB16 instruction</td>
</tr>
<tr>
<td>__UADD8</td>
<td>Inserts a UADD8 instruction</td>
</tr>
<tr>
<td>__UADD16</td>
<td>Inserts a UADD16 instruction</td>
</tr>
<tr>
<td>__UASX</td>
<td>Inserts a UASX instruction</td>
</tr>
<tr>
<td>__UHADD8</td>
<td>Inserts a UHADD8 instruction</td>
</tr>
<tr>
<td>__UHADD16</td>
<td>Inserts a UHADD16 instruction</td>
</tr>
<tr>
<td>__UHASX</td>
<td>Inserts a UHASX instruction</td>
</tr>
<tr>
<td>__UHSAX</td>
<td>Inserts a UHSAX instruction</td>
</tr>
<tr>
<td>__UHSUB8</td>
<td>Inserts a UHSUB8 instruction</td>
</tr>
<tr>
<td>__UHSUB16</td>
<td>Inserts a UHSUB16 instruction</td>
</tr>
<tr>
<td>__UMAAL</td>
<td>Inserts a UMAAL instruction</td>
</tr>
<tr>
<td>__UQADD8</td>
<td>Inserts a UQADD8 instruction</td>
</tr>
<tr>
<td>__UQADD16</td>
<td>Inserts a UQADD16 instruction</td>
</tr>
<tr>
<td>__UQASX</td>
<td>Inserts a UQASX instruction</td>
</tr>
<tr>
<td>__UQASX</td>
<td>Inserts a UQASX instruction</td>
</tr>
<tr>
<td>__UQSUB8</td>
<td>Inserts a UQSUB8 instruction</td>
</tr>
<tr>
<td>__UQSUB16</td>
<td>Inserts a UQSUB16 instruction</td>
</tr>
<tr>
<td>__USAD8</td>
<td>Inserts a USAD8 instruction</td>
</tr>
</tbody>
</table>

Table 36: Intrinsic functions summary (Continued)
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
```

- **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:

```c
typedef struct {
    float32x2_t val[4];
} float32x2x4_t;
```

---

**Descriptions of intrinsic functions**

This section gives reference information about each intrinsic function.

---

**__CLREX**

**Syntax**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
void __DMB(void);
```

**Description**

Inserts a `DMB` instruction. This intrinsic function requires an ARMv6M architecture, or an ARMv7 architecture or higher.

### __DSB

**Syntax**

```c
void __DSB(void);
```

**Description**

Inserts a `DSB` instruction. This intrinsic function requires an ARMv6M architecture, or an ARMv7 architecture or higher.

### __enable_fiq

**Syntax**

```c
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.
Descriptions of intrinsic functions

__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  
unsigned long __get_CONTROL(void);

Description  
Returns the value of the CONTROL register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__get_CPSR

Syntax  
unsigned long __get_CPSR(void);

Description  
Returns the value of the ARM CPSR (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.
Part 2. Reference information

Intrinsic functions

__get_FAULTMASK

Syntax
unsigned long __get_FAULTMASK(void);

Description
Returns 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.

__get_FPSCR

Syntax
unsigned long __get_FPSCR(void);

Description
Returns the value of FPSCR (floating-point status and control register). This intrinsic function is only available for devices with a VFP coprocessor.

__get_interrupt_state

Syntax
__istate_t __get_interrupt_state(void);

Description
Returns the global interrupt state. The return value can be used as an argument to the __set_interrupt_state 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 --aeabi 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.
Intrinsic functions

__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
void __ISB(void);
Description
Inserts an ISB instruction. This intrinsic function requires an ARMv6M architecture, or an ARMv7 architecture or higher.

__LDC
__LDCL
__LDC2
__LDC2L
Syntax
void __LDCxxx(__ul coproc, __ul CRn, __ul const *src);
Descriptions of intrinsic functions

Parameters

- **coproc**: The coprocessor number 0..15.
- **CRn**: The coprocessor register to load.
- **src**: A pointer to the data to load.

Description

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 *coproc* and CRn will be encoded in the instruction and must therefore be constants.

The intrinsic functions __LDC and __LDCL require architecture ARMv4 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.

The intrinsic functions __LDC2 and __LDC2L require architecture ARMv5 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.

---

__LDC_noidx__

__LDCL_noidx__

__LDC2_noidx__

__LDC2L_noidx__

**Syntax**

```c
void __LDCxxx_noidx(__ul coproc, __ul CRn, __ul const *src, __ul option);
```

**Parameters**

- **coproc**: The coprocessor number 0..15.
- **CRn**: The coprocessor register to load.
- **src**: A pointer to the data to load.
- **option**: Additional coprocessor option 0..255.

**Description**

Inserts the coprocessor load instruction LDC, or one of its variants. A value will be loaded into a coprocessor register. The parameters *coproc*, CRn, and *option* will be encoded in the instruction and must therefore be constants.

The intrinsic functions __LDC_noidx and __LDCL_noidx require architecture ARMv4 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.
Part 2. Reference information

Intrinsic functions

The intrinsic functions __LDC2_noidx and __LDC2L_noidx require architecture ARMv5 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.

__LDREX

__LDREXB

__LDREXD

__LDREXH

Syntax

unsigned long __LDREX(unsigned long *);  
unsigned char __LDREXB(unsigned char *);  
unsigned long long __LDREXD(unsigned long long *);  
unsigned short __LDREXH(unsigned short *);

Description

Inserts the specified instruction.

The __LDREX intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv6T2 or higher for Thumb mode.

The __LDREXB and the __LDREXH intrinsic functions require architecture ARMv6K or ARMv7 for ARM mode, and ARMv7 for Thumb mode.

The __LDREX intrinsic function requires architecture ARMv6K or ARMv7 for ARM mode, and ARMv7 but not ARMv7-M for Thumb mode.

__MCR

__MCR2

Syntax

void __MCR(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul CRm, __ul opcode_2);  
void __MCR2(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul CRm, __ul opcode_2);

Parameters

coproc
opcode_1

The coprocessor number 0..15.
Coprocessor-specific operation code.
Descriptions of intrinsic functions

- **__MCR**

  **Syntax**

  ```c
  unsigned long __MRC(__ul coproc, __ul opcode_1, __ul CRn, __ul CRm, __ul opcode_2);
  ```

  **Parameters**

  - `coproc` The coprocessor number 0..15.
  - `opcode_1` Coprocessor-specific operation code.
  - `CRn` The coprocessor register to write to.
  - `CRm` Additional coprocessor register; set to zero if not used.
  - `opcode_2` Additional coprocessor-specific operation code; set to zero if not used.

  **Description**

  Inserts a coprocessor write instruction (MCR or MCR2). A value will be written to a coprocessor register. The parameters `coproc`, `opcode_1`, `CRn`, `CRm`, and `opcode_2` will be encoded in the instruction and must therefore be constants.

  The intrinsic function `__MCR` requires either ARM mode, or an ARMv6T2 or higher for Thumb mode.

  The intrinsic function `__MCR2` requires an ARMv5T architecture or higher for ARM mode, or ARMv6T2 or higher for Thumb mode.

- **__MRC2**

  **Syntax**

  ```c
  unsigned long __MRC2(__ul coproc, __ul opcode_1, __ul CRn, __ul CRm, __ul opcode_2);
  ```

  **Parameters**

  - `coproc` The coprocessor number 0..15.
  - `opcode_1` Coprocessor-specific operation code.
  - `CRn` The coprocessor register to write to.
  - `CRm` Additional coprocessor register; set to zero if not used.
  - `opcode_2` Additional coprocessor-specific operation code; set to zero if not used.

  **Description**

  Inserts a coprocessor read instruction (MRC or MRC2). Returns the value of the specified coprocessor register. The parameters `coproc`, `opcode_1`, `CRn`, `CRm`, and `opcode_2` will be encoded in the instruction and must therefore be constants.

  The intrinsic function `__MRC` requires either ARM mode, or an ARMv6T2 or higher for Thumb mode.
Intrinsic functions

The intrinsic function __MRC2 requires an ARMv5T architecture or higher for ARM mode, or ARMv6T2 or higher for Thumb mode.

__no_operation
Syntax
void __no_operation(void);
Description
Inserts a NOP instruction.

__PKHBT
Syntax
unsigned long __PKHBT(unsigned long x, unsigned long y, unsigned long count);
Parameters
x First operand.
y Second operand, optionally shifted left.
count Shift count 0–31, where 0 means no shift.
Description
Inserts a PKHBT 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 x, unsigned long y, unsigned long count);
Parameters
x First operand.
y Second operand, optionally shifted right (arithmetic shift).
count Shift count 0–32, where 0 means no shift.
Description
Inserts a PKHTB 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.

___PLD

___PLDW

Syntax

void ___PLD(void const *);
void ___PLDW(void const *);

Description

Inserts a preload data instruction (PLD or PLDW).

The intrinsic function ___PLD requires an ARMv7 architecture. ___PLDW requires an ARMv7 architecture with MP extensions (for example Cortex-A5).

___PLI

Syntax

void ___PLI(void const *);

Description

Inserts a PLI instruction.

This intrinsic function requires an ARM v7 architecture.

___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
unsigned long __Qxxx(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.

__QCFlag

Syntax
unsigned long __QCFlag(void);

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
signed long __QDOUBLE(signed long);

Description
Inserts an instruction QADD Rd, Rs, Rs for a source register Rs, and a destination register Rd.
This intrinsic function requires architecture ARMv5E or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7E-M for Thumb mode.
Descriptions of intrinsic functions

__QFlag

Syntax

int __QFlag(void);

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

unsigned long __RBIT(unsigned long);

Description

Inserts an RBIT instruction, which reverses the bit order in a 32-bit register. This intrinsic function requires architecture ARMv6T2 or higher.

__reset_Q_flag

Syntax

void __reset_Q_flag(void);

Description

Clears the Q 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

void __reset_QC_flag(void);

Description

Clears 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).
Intrinsic functions

__REV

__REV16

__REVSH

Syntax

unsigned long __REV(unsigned long);
unsigned long __REV16(unsigned long);
signed long __REVSH(short);

Description

Inserts the specified instruction.

These intrinsic functions require architecture ARMv6 or higher.

__SADD8

__SADD16

__SASX

__SSAX

__SSUB8

__SSUB16

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: void __set_CPSR(unsigned long);
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: void __set_FAULTMASK(unsigned long);
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: void __set_FPSCR(unsigned long);
Description: Sets the value of FPSCR (floating-point status and control register) This intrinsic function is only available for devices with a VFP coprocessor.
Intrinsic functions

__set_interrupt_state
Syntax
void __set_interrupt_state(__istate_t);
Description
Restores the interrupt state to a value previously returned by the
__get_interrupt_state function.
For information about the __istate_t type, see __get_interrupt_state, page 355.

__set_LR
Syntax
void __set_LR(unsigned long);
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
__SHADD16
__SHASX
__SHSAX
__SHSUB8
__SHSUB16
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 __SMLxxxx(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 __SMLxxxx(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);

Description
Inserts the specified instruction.

__SMLALD
__SMLALDX
__SMLSLD
__SMLSLDX

Syntax
unsigned long 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.
<table>
<thead>
<tr>
<th>Intrinsic functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>__SMMLA</td>
</tr>
<tr>
<td>__SMMLAR</td>
</tr>
<tr>
<td>__SMMLS</td>
</tr>
<tr>
<td>__SMMLSR</td>
</tr>
</tbody>
</table>

**Syntax**

```c
unsigned long __SMMLxxx(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.

| __SMMUL             |
| __SMMULR            |

**Syntax**

```c
unsigned long __SMMULxxx(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.

| __SMUAD             |
| __SMUADX            |
| __SMUSD             |
| __SMUSDX            |

**Syntax**

```c
unsigned long __SMUxxx(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.

**__SMUL**

**Syntax**

`signed long __SMUL(signed short, signed short);`

**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**

**__SMULLT**

**__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, \texttt{__SSAT(x \ll 3,11)} compiles to \texttt{SSAT Rd,#11,Rn,LSL #3}, where the value of \texttt{x} has been placed in register \texttt{Rn} and the return value of \texttt{__SSAT} will be placed in register \texttt{Rd}.

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7-M for Thumb mode.

\section*{__SSAT16}

\begin{description}
\item[Syntax] \texttt{unsigned long __SSAT16(unsigned long, unsigned long);}
\item[Description] Inserts an \texttt{SSAT16} instruction.
\end{description}

This intrinsic function requires architecture ARMv6 or higher for ARM mode, and ARMv7-A, ARMv7-R, or ARMv7-E for Thumb mode.

\section*{__STC}

\section*{__STCL}

\section*{__STC2}

\section*{__STC2L}

\begin{description}
\item[Syntax] \texttt{void __STCxxx(__ul coproc, __ul CRn, __ul const *dst);} \texttt{)}
\item[Parameters]
\begin{itemize}
\item \texttt{coproc} The coprocessor number \texttt{0..15}.
\item \texttt{CRn} The coprocessor register to load.
\item \texttt{dst} A pointer to the destination.
\end{itemize}
\item[Description] Inserts the coprocessor store instruction \texttt{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 \texttt{coproc} and \texttt{CRn} will be encoded in the instruction and must therefore be constants.

The intrinsic functions \texttt{__STC} and \texttt{__STCL} require architecture ARMv4 or higher for ARM mode, and ARMv6T2 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

- coproc: The coprocessor number 0..15.
- CRn: The coprocessor register to load.
- dst: A pointer to the destination.
- option: 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

Syntax

unsigned long __STREX(unsigned long, unsigned long *);

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.

### __STREXB

Syntax

unsigned long __STREXB(unsigned char, unsigned char *);

### __STREXD

Syntax

unsigned long __STREXD(unsigned long long, unsigned long long *);

### __STREXH

Syntax

unsigned long __STREXH(unsigned short, unsigned short *);

Description

Inserts the specified instruction.

The __STREXH intrinsic function requires 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

Syntax

unsigned long __SWP(unsigned long, unsigned long *);

### __SWPB

Syntax

unsigned long __SWPB(unsigned char, unsigned char *);

Description

Inserts the specified instruction.

These intrinsic functions require ARM mode.
Descriptions of intrinsic functions

__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.
Intrinsic functions

__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.
Descriptions of intrinsic functions

__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.
<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__WFI</td>
<td>void long __xxx(void);</td>
<td>Inserts the specified instruction.</td>
</tr>
<tr>
<td>__WFE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__YIELD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 *Description of predefined preprocessor symbols*, page 382.

- **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 237.

- **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 387.

- **Preprocessor output**
  Use the option `--preprocess` to direct preprocessor output to a named file, see `--preprocess`, page 260.

To specify a path for an include file, use forward slashes:

```c
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```c
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.
### Description of predefined preprocessor symbols

This section lists and describes the preprocessor symbols.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AAPCS</strong></td>
<td>An integer that is set based on the --aapcs option. The symbol is set to 1 if the AAPCS base standard is the selected calling convention (--aapcs=std). The symbol is undefined for other calling conventions.</td>
<td>--aapcs, page 233.</td>
</tr>
<tr>
<td><strong>AAPCS_VFP</strong></td>
<td>An integer that is set based on the --aapcs option. The symbol is set to 1 if the VFP variant of AAPCS is the selected calling convention (--aapcs=vfp). The symbol is undefined for other calling conventions.</td>
<td>--aapcs, page 233.</td>
</tr>
<tr>
<td><strong>ARM_ADVANCED_SIMD</strong></td>
<td>An integer that is set based on the --cpu 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.</td>
<td>--cpu, page 235.</td>
</tr>
<tr>
<td><strong>ARM_MEDIA</strong></td>
<td>An integer that is set based on the --cpu 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.</td>
<td>--cpu, page 235.</td>
</tr>
</tbody>
</table>
**__ARM_PROFILE_M__**

**Description**
An integer that is set based on the `--cpu` option. The symbol is set to 1 if the selected processor architecture is a profile M core. The symbol is undefined for other cores.

**See also**
`--cpu`, page 235.

**__ARMVFP__**

**Description**
An integer that reflects the `--fpu` option and is defined to `__ARMVFPV2__`, `__ARMVFPV3__`, or `__ARMVFPV4__`. These symbolic names can be used when testing the `__ARMVFP__` symbol. If VFP code generation is disabled (default), the symbol will be undefined.

**See also**
`--fpu`, page 246.

**__ARMVFP_D16__**

**Description**
An integer that is set based on the `--fpu` 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.

**See also**
`--fpu`, page 246.

**__ARMVFP_FP16__**

**Description**
An integer that is set based on the `--fpu` option. The symbol is set to 1 if the selected FPU only supports 16-bit floating-point numbers. Otherwise, the symbol is undefined.

**See also**
`--fpu`, page 246.

**__ARMVFP_SP__**

**Description**
An integer that is set based on the `--fpu` option. The symbol is set to 1 if the selected FPU only supports 16-bit single-precision. Otherwise, the symbol is undefined.

**See also**
`--fpu`, page 246.
Description of predefined preprocessor symbols

__BASE_FILE__

Description
A string that identifies the name of the base source file (that is, not the header file), being compiled.

See also
See also __FILE__, page 385, and --no_path_in_file_macros, page 254.

__BUILD_NUMBER__

Description
A unique integer that identifies the build number of the compiler currently in use.

__CORE__

Description
An integer that identifies the processor architecture in use. The symbol reflects the --core option and is defined to __ARM4TM__, __ARM5__, __ARM5E__, __ARM6__, __ARM6K__, __ARM6SM__, __ARM7__, __ARM7EM__, __ARM7A__, or __ARM7B__.

These symbolic names can be used when testing the __CORE__ symbol.

__cplusplus__

Description
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 #ifdef 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.

This symbol is required by Standard C.

__CPU_MODE__

Description
An integer that reflects the selected CPU mode and is defined to 1 for Thumb and 2 for ARM.

__DATE__

Description
A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Oct 30 2010"

This symbol is required by Standard C.
__DOUBLE__
Description: An integer that identifies the size of the data type double. The symbol is defined to 64.

__embedded_cplusplus
Description: 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 #ifdef 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.
This symbol is required by Standard C.

__FILE__
Description: A string that identifies the name of the file being compiled, which can be both the base source file and any included header file.
This symbol is required by Standard C.
See also: See also __BASE_FILE__, page 384, and --no_path_in_file_macros, page 254.

__func__
Description: 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.
This symbol is required by Standard C.
See also: -e, page 242. See also __PRETTY_FUNCTION__, page 386.

__FUNCTION__
Description: 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 also: -e, page 242. See also __PRETTY_FUNCTION__, page 386.
Description of predefined preprocessor symbols

__IAR_SYSTEMS_ICC__
Description
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 #ifdef to detect whether the code was compiled by a compiler from IAR Systems.

__ICCARM__
Description
An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for
ARM.

__LINE__
Description
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.
This symbol is required by Standard C.

__LITTLE_ENDIAN__
Description
An integer that reflects the --endian 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.

__PRETTY_FUNCTION__
Description
A predefined string identifier that is initialized with the function name, including
parameter types and return type, of the function in which the symbol is used, for
example "void func(char)". This symbol is useful for assertions and other trace
utilities. The symbol requires that language extensions are enabled.
See also
-e, page 242. See also __func__, page 385.

__ROPI__
Description
An integer that is defined when the --ropi compiler option is used.
See also
--ropi, page 262.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RWPI</strong></td>
<td>An integer that is defined when the --rwpi compiler option is used.</td>
<td>--rwpi, page 263</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>An integer that is set to 1, which means the compiler adheres to Standard C.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This symbol can be tested with #ifdef to detect whether the compiler in use adheres to Standard C.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This symbol is required by Standard C.</td>
<td></td>
</tr>
<tr>
<td><strong>STDC_VERSION</strong></td>
<td>An integer that identifies the version of the C standard in use.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This symbol is required by Standard C.</td>
<td></td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>A string that identifies the time of compilation in the form &quot;hh:mm:ss&quot;.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This symbol is required by Standard C.</td>
<td></td>
</tr>
<tr>
<td><strong>VER</strong></td>
<td>An integer that identifies the version number of the IAR compiler in use.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For example, version 5.11.3 is returned as 5011003.</td>
<td></td>
</tr>
</tbody>
</table>

**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 130.

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 100. 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 \texttt{cos}, \texttt{sin}, \texttt{tan}, and \texttt{pow} is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named \texttt{__iar\_xxx\_accuratef} for float variants of the functions and \texttt{__iar\_xxx\_accuratel} for long double variants of the functions, and where \texttt{xxx} is \texttt{cos, sin, etc}.

To use any of these more accurate versions, use the \texttt{--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—\texttt{malloc, free, realloc, calloc, and the C++ operators new and delete}
- Locale functions—\texttt{localeconv, setlocale}
- Multibyte functions—\texttt{mbrlen, mbrtowc, mbstowc, mbtowc, wcrtomb, wcsrtomb, wctomb}
- Rand functions—\texttt{rand, srand}
- Time functions—\texttt{asctime, localtime, gmtime, mktime}
- The miscellaneous functions \texttt{atexit, strerror, strtok}
- Functions that use files or the heap in some way. This includes \texttt{printf, sprintf, scanf, sscanf, getchar, and putchar.}

Functions that can set \texttt{errno} are not reentrant, because an \texttt{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 LONGJUMP 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 for Standard C* 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 395.

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:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert.h</td>
<td>Enforcing assertions when functions execute</td>
</tr>
<tr>
<td>complex.h</td>
<td>Computing common complex mathematical functions</td>
</tr>
<tr>
<td>ctype.h</td>
<td>Classifying characters</td>
</tr>
<tr>
<td>errno.h</td>
<td>Testing error codes reported by library functions</td>
</tr>
</tbody>
</table>

*Table 37: Traditional Standard C header files—DLIB*
<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>fenv.h</td>
<td>Floating-point exception flags</td>
</tr>
<tr>
<td>float.h</td>
<td>Testing floating-point type properties</td>
</tr>
<tr>
<td>inttypes.h</td>
<td>Defining formatters for all types defined in stdint.h</td>
</tr>
<tr>
<td>iso646.h</td>
<td>Using Amendment 1—iso646.h standard header</td>
</tr>
<tr>
<td>limits.h</td>
<td>Testing integer type properties</td>
</tr>
<tr>
<td>locale.h</td>
<td>Adapting to different cultural conventions</td>
</tr>
<tr>
<td>math.h</td>
<td>Computing common mathematical functions</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>Executing non-local goto statements</td>
</tr>
<tr>
<td>signal.h</td>
<td>Controlling various exceptional conditions</td>
</tr>
<tr>
<td>stdarg.h</td>
<td>Accessing a varying number of arguments</td>
</tr>
<tr>
<td>stdbool.h</td>
<td>Adds support for the bool data type in C.</td>
</tr>
<tr>
<td>stddef.h</td>
<td>Defining several useful types and macros</td>
</tr>
<tr>
<td>stdint.h</td>
<td>Providing integer characteristics</td>
</tr>
<tr>
<td>stdio.h</td>
<td>Performing input and output</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>Performing a variety of operations</td>
</tr>
<tr>
<td>string.h</td>
<td>Manipulating several kinds of strings</td>
</tr>
<tr>
<td>tgmath.h</td>
<td>Type-generic mathematical functions</td>
</tr>
<tr>
<td>time.h</td>
<td>Converting between various time and date formats</td>
</tr>
<tr>
<td>uchar.h</td>
<td>Unicode functionality (IAR extension to Standard C)</td>
</tr>
<tr>
<td>wchar.h</td>
<td>Support for wide characters</td>
</tr>
<tr>
<td>wctype.h</td>
<td>Classifying wide characters</td>
</tr>
</tbody>
</table>

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++:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex</td>
<td>Defining a class that supports complex arithmetic</td>
</tr>
<tr>
<td>exception</td>
<td>Defining several functions that control exception handling; only usable in C++</td>
</tr>
<tr>
<td>fstream</td>
<td>Defining several I/O stream classes that manipulate external files</td>
</tr>
<tr>
<td>iomanip</td>
<td>Declaring several I/O stream manipulators that take an argument</td>
</tr>
<tr>
<td>ios</td>
<td>Defining the class that serves as the base for many I/O streams classes</td>
</tr>
<tr>
<td>iosfwd</td>
<td>Declaring several I/O stream classes before they are necessarily defined</td>
</tr>
<tr>
<td>iostream</td>
<td>Declaring the I/O stream objects that manipulate the standard streams</td>
</tr>
<tr>
<td>istream</td>
<td>Defining the class that performs extractions</td>
</tr>
<tr>
<td>limits</td>
<td>Defining numerical values; only usable in C++</td>
</tr>
<tr>
<td>locale</td>
<td>Adapting to different cultural conventions; only usable in C++</td>
</tr>
<tr>
<td>new</td>
<td>Declaring several functions that allocate and free storage</td>
</tr>
<tr>
<td>ostream</td>
<td>Defining the class that performs insertions</td>
</tr>
<tr>
<td>sstream</td>
<td>Defining several I/O stream classes that manipulate string containers</td>
</tr>
<tr>
<td>stdexcept</td>
<td>Defining several classes useful for reporting exceptions; only usable in C++</td>
</tr>
<tr>
<td>streambuf</td>
<td>Defining classes that buffer I/O stream operations</td>
</tr>
<tr>
<td>string</td>
<td>Defining a class that implements a string container</td>
</tr>
<tr>
<td>stringstream</td>
<td>Defining several I/O stream classes that manipulate in-memory character sequences</td>
</tr>
<tr>
<td>typeinfo</td>
<td>Defining type information support; only usable in C++</td>
</tr>
</tbody>
</table>

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++:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>Defines several common operations on sequences</td>
</tr>
<tr>
<td>bitset</td>
<td>Defining a container with fixed-sized sequences of bits; only usable in C++</td>
</tr>
<tr>
<td>deque</td>
<td>A deque sequence container</td>
</tr>
<tr>
<td>functional</td>
<td>Defines several function objects</td>
</tr>
</tbody>
</table>

Table 39: Standard template library header files
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:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassert</td>
<td>Enforcing assertions when functions execute</td>
</tr>
<tr>
<td>ccomplex</td>
<td>Computing common complex mathematical functions</td>
</tr>
<tr>
<td>cctype</td>
<td>Classifying characters</td>
</tr>
<tr>
<td>cerrno</td>
<td>Testing error codes reported by library functions</td>
</tr>
<tr>
<td>cfenv</td>
<td>Floating-point exception flags</td>
</tr>
<tr>
<td>cfloat</td>
<td>Testing floating-point type properties</td>
</tr>
<tr>
<td>cinttypes</td>
<td>Defining formatters for all types defined in stdint.h</td>
</tr>
<tr>
<td>ciso646</td>
<td>Using Amendment I—iso646.h standard header</td>
</tr>
<tr>
<td>climits</td>
<td>Testing integer type properties</td>
</tr>
<tr>
<td>clocale</td>
<td>Adapting to different cultural conventions</td>
</tr>
</tbody>
</table>

Table 40: New Standard C header files—DLIB

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:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash_map</td>
<td>A map associative container, based on a hash algorithm</td>
</tr>
<tr>
<td>hash_set</td>
<td>A set associative container, based on a hash algorithm</td>
</tr>
<tr>
<td>iterator</td>
<td>Defines common iterators, and operations on iterators</td>
</tr>
<tr>
<td>list</td>
<td>A doubly-linked list sequence container</td>
</tr>
<tr>
<td>map</td>
<td>A map associative container</td>
</tr>
<tr>
<td>memory</td>
<td>Defines facilities for managing memory</td>
</tr>
<tr>
<td>numeric</td>
<td>Performs generalized numeric operations on sequences</td>
</tr>
<tr>
<td>queue</td>
<td>A queue sequence container</td>
</tr>
<tr>
<td>set</td>
<td>A set associative container</td>
</tr>
<tr>
<td>slist</td>
<td>A singly-linked list sequence container</td>
</tr>
<tr>
<td>stack</td>
<td>A stack sequence container</td>
</tr>
<tr>
<td>utility</td>
<td>Defines several utility components</td>
</tr>
<tr>
<td>valarray</td>
<td>Defining varying length array container; only usable in C++</td>
</tr>
<tr>
<td>vector</td>
<td>A vector sequence container</td>
</tr>
</tbody>
</table>

Table 39: Standard template library header files (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.
These functions provide additional I/O functionality:

- `fdopen` Opens a file based on a low-level file descriptor.
- `fileno` Gets the low-level file descriptor from the file descriptor (FILE*).
- `__gets` Corresponds to `fgets` on stdin.
- `getw` Gets a wchar_t character from stdin.
- `putw` Puts a wchar_t character to stdout.
- `__ungetchar` Corresponds to `ungetc` on stdout.
- `__write_array` Corresponds to `fwrite` on stdout.

These are the additional functions defined in `string.h`:

- `strdup` Duplicates a string on the heap.
- `strcasestr` Compares strings case-insensitive.
- `strncasecmp` Compares strings case-insensitive and bounded.
- `strnlen` Bounded string length.

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 127.

**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 70.

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.
Defining memories and regions

Structural configuration

- 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 400.

- 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 401.

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 403.

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 419.

Parameters

- `size_expr` Specifies how many units the memory space contains; always counted from address zero.
- `bitsize_expr` Specifies how many bits each unit contains.
- `bytesize_expr` Specifies how many bytes each unit contains. Each byte contains 8 bits.
Part 2. Reference information

The linker configuration file

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. 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 `unit-size` is given, the unit contains 8 bits.

Example
/* Declare the memory space Mem of four Gigabytes */
define memory Mem with size = 4G;

Define region directive

Syntax
define region name = region-expr;
where `region-expr` is a region expression, see also `Regions`, page 401.

Parameters
name
The name of the region.

Description
The `define region` 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
/* Define the 0x10000-byte code region ROM located at address 0x10000 in memory Mem */
define region ROM = Mem:[from 0x10000 size 0x10000];

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

\[
\text{[memory-name: ] [from expr (to expr | size expr )]}
\]

\[
\text{[repeat expr [displacement expr]]}
\]

where expr is an expression, see Expressions, page 419.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory-name</td>
<td>The name of the memory space in which the region literal will be located. If there is only one memory, the name is optional.</td>
</tr>
<tr>
<td>from expr</td>
<td>expr is the start address of the memory range (inclusive).</td>
</tr>
<tr>
<td>to expr</td>
<td>expr is the end address of the memory range (inclusive).</td>
</tr>
<tr>
<td>size expr</td>
<td>expr is the size of the memory range.</td>
</tr>
<tr>
<td>repeat expr</td>
<td>expr defines several ranges in the same memory for the region literal.</td>
</tr>
<tr>
<td>displacement</td>
<td>expr is the displacement from the previous range start in the repeat sequence. Default displacement is the same value as the range size.</td>
</tr>
</tbody>
</table>

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.
Part 2. Reference information

The linker configuration file

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 401, and Region expression, page 403.

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 401
where region-literal is a region literal, see Region literal, page 402
and where empty-region is an empty region, see Empty region, page 404.

**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 (\(\cup\)), intersection (\(\cap\)), and difference (\(\setminus\)). 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:

- \(A \cup B\): all elements in either set A or set B
- \(A \cap B\): 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**

```
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. */
```
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 412 and `Place in directive`, page 413.

- 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 405, and `Define overlay directive`, page 407.

- 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 408 and `Do not initialize directive`, page 410.

- 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 411.

Define block directive

**Syntax**

```
define [movable] block name
  [ with param, param... ]
  
  { extended-selectors
    } 
  
  [except 
    { section_selectors
    }]
```

The linker configuration file

See also

Region expression, page 403.
where \( \text{param} \) can be one of:

- \( \text{size} = \text{expr} \)
- \( \text{maximum size} = \text{expr} \)
- \( \text{alignment} = \text{expr} \)
- \( \text{fixed order} \)
- \( \text{static base} \ [\text{name}] \)
- \( \text{static base} \)

and where the rest of the directive selects sections to include in the block, see Section selection, page 413.

**Parameters**

- **name**
  - The name of the block to be defined.

- **size**
  - Customizes the size of the block. By default, the size of a block is the sum of its parts dependent of its contents.

- **maximum size**
  - Specifies an upper limit for the size of the block. An error is generated if the sections in the block do not fit.

- **alignment**
  - 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.

- **fixed order**
  - Places sections in fixed order; if not specified, the order of the sections will be arbitrary.

- **static base name**
  - Specifies that the static base with the specified name will be at the start of the block when the application executes.

- **static base**
  - Specifies that the static base will be at the start of the block when the application executes.

**Description**

The `block` 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.

Movable 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.

**Example**

/* Create a 0x1000-byte block for the heap */
define block HEAP with size = 0x1000, alignment = 8 { };
The linker configuration file

See also

Interaction between the tools and your application, page 186. See Define overlay directive, page 407 for an Accessing example.

Define overlay directive

Syntax

```
define overlay name [ with param, param... ]
{ extended-selectors;
} [except
    { section_selectors
    }];
```

For information about extended selectors and except clauses, see Section selection, page 413.

Parameters

- **name**: The name of the overlay.
- **size**: Customizes the size of the overlay. By default, the size of a overlay is the sum of its parts dependent of its contents.
- **maximum size**: Specifies an upper limit for the size of the overlay. An error is generated if the sections in the overlay do not fit.
- **alignment**: 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.
- **fixed order**: 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 91.

**Initialize directive**

**Syntax**

```c
initialize { by copy | manually }
[ with packing = algorithm ]
{ section-selectors }
[except {
  section_selectors
}]
```

where the rest of the directive selects sections to include in the block. See *Section selection*, page 413.

**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.
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 the decompressors for the other methods. Approximately 9 Kbytes of stack space is needed for bwt and 3.5 Kbytes for lzw.

algorithm

Specifies how to handle the initializers. Choose between:

- none - Disables compression of the selected section contents. This is the default method for initialize manually.
- zeros - Compresses consecutive bytes with the value zero.
- packbits - Compresses with the PackBits algorithm. This method generates good results for data with many consecutive bytes of the same value.
- lz77 - Compresses with the Lempel-Ziv-77 algorithm. This method handles a larger variety of inputs well, but has a slightly larger decompressor.
- bwt - Compresses with the Burrows-Wheeler algorithm. This method improves the packbits method by transforming blocks of data before they are compressed.
- lzw - Compresses with the Lempel-Ziv-Welch algorithm. This method uses a dictionary to store byte patterns in the data.
- auto - Similar to smallest, but ILINK chooses between none, packbits, and lz77. This is the default method for initialize by copy.
- smallest - ILINK estimates the resulting size using each packing method (except for auto), and then chooses the packing method that produces the smallest estimated size. Note that the size of the decompressor is also included.

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 the decompressors for the other methods. 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 407.

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 76, and Do not initialize directive, page 410.

Do not initialize directive

Syntax

do not initialize
{
section-selectors
}
[except
{
section-selectors
}];
The linker configuration file

Part 2. Reference information

For information about extended selectors and except clauses, see Section selection, page 413.

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 76, and Initialize directive, page 408.

Keep directive

Syntax

keep
{
    section-selectors
}
[except
{
    section-selectors
}];

For information about extended selectors and except clauses, see Section selection, page 413.

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 }
```

Parameters

- **memory: expr**
  A specific address in a specific memory. The address must be available in the supplied memory defined by the `define memory` directive. The memory specifier is optional if there is only one memory.

- **start of region_expr**
  A region expression that results in a single-internal region. The start of the interval is used.

- **end of region_expr**
  A region expression that results in a single-internal region. The end of the interval 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 `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 413.
Place in directive

Syntax

```plaintext
[ "name": ]
place in region-expr
{
    extended-selectors
}
[except{
    section-selectors
}];
```

where `region-expr` is a region expression, see also `Regions`, page 401.

and where the rest of the directive selects sections to include in the block. See `Section selection`, page 413.

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

```plaintext
/* Place the read-only sections in the code_region */
"ROM": place in ROM { readonly };
```

See also

`Place at directive`, page 412.

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

```plaintext
( [ 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**
  - The name of the block.
- **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 `objectname(libraryname)`. 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.
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.

<table>
<thead>
<tr>
<th>Selector 1</th>
<th>Selector 2</th>
<th>More specific</th>
</tr>
</thead>
<tbody>
<tr>
<td>section &quot;foo*&quot;</td>
<td>section &quot;f*&quot;</td>
<td>Selector 1</td>
</tr>
<tr>
<td>section &quot;x*&quot;</td>
<td>section &quot;f*&quot;</td>
<td>Neither</td>
</tr>
<tr>
<td>no code section &quot;f*&quot;</td>
<td>ro section &quot;f*&quot;</td>
<td>Selector 1</td>
</tr>
<tr>
<td>init_array</td>
<td>ro section &quot;xx*&quot;</td>
<td>Selector 1</td>
</tr>
<tr>
<td>section &quot;.intvec&quot;</td>
<td>ro section &quot;.int&quot;</td>
<td>Selector 1</td>
</tr>
<tr>
<td>section &quot;.intvec&quot;</td>
<td>object &quot;foo.o&quot;</td>
<td>Neither</td>
</tr>
</tbody>
</table>

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


Extended-selectors

Syntax

```
( [ extended-selector ][, extended-selector... ] )
```

where extended-selector is:

```
[ first | last ]{ section-selector |
    block name [ inline-block-def ] |
    overlay name }
```

where inline-block-def is:

```
[ block-params ] extended-selectors
```

Parameters

- **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 `section-selector` does, `extended-selector` 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 `inline` definition of a block. This means that you can get more precise control over section placement.

Example

```c
#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 */
```

or, equivalently using an inline definition of the block `First`:

```c
#define block Table ( first block First { section .first });
```
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 418, and Export directive, page 418.

- Use expressions and numbers
  In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, etc. See Expressions, page 419.

Check that directive

**Syntax**
```
check that expression;
```

**Parameters**

- `expression` A boolean expression.

**Description**

You can use the check that directive to compare the results of stack usage analysis against the sizes of blocks and regions. If the expression evaluates to zero, an error is emitted.

Three extra operators are available for use only in check that expressions:

- `maxstack(category)` The stack depth of the deepest call chain for any call graph root function in the category.
- `totalstack(category)` The sum of the stack depths of the deepest call chains for each call graph root function in the category.
- `size(block)` The size of the block.

**Example**
```
check that maxstack("Program entry")
+ totalstack("interrupt")
+ 1K <= size(block CSTACK);
```

**See also**
Stack usage analysis, page 78.
Define symbol directive

Syntax

```
define [ exported ] symbol name = expr;
```

Parameters

- `exported` (Exports the symbol to be usable by the executable image.)
- `name` (The name of the symbol.)
- `expr` (The symbol value.)

Description

The `define` symbol 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 `--config_def` outside of the configuration file.

The `define exported symbol` variant of this directive is a shortcut for using the directive `define symbol` in combination with the `export symbol` directive. On the command line this would require both a `--config_def` option and a `--define_symbol` option to achieve the same effect.

Note:

- A symbol cannot be redefined
- Symbols that are either prefixed by `_X`, where `X` is a capital letter, or that contain `__` (double underscore) are reserved for toolset vendors.

Example

```
/* Define the symbol my_symbol with the value 4 */
define symbol my_symbol = 4;
```

See also

Export directive, page 418 and Interaction between ILINK and the application, page 94.

Export directive

Syntax

```
export symbol name;
```

Parameters

- `name` (The name of the symbol.)

Description

The `export` 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.
Expressions

Example

/* Define the symbol my_symbol to be exported */
export symbol my_symbol;

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 420

where symbol is a defined symbol, see Define symbol directive, page 418 and --config_def, page 274

and where func-operator is one of these function-like operators:

minimum(expr, expr) Returns the smallest of the two parameters.
maximum(expr, expr) Returns the largest of the two parameters.
isempty(r) Returns True if the region is empty, otherwise False.
isdefinedsymbol(expr-symbol) Returns True if the expression symbol is defined, otherwise False.
start(r) Returns the lowest address in the region.
end(r) Returns the highest address in the region.
size(r) Returns the size of the complete region.

where expr is an expression, and r is a region expression, see Region expression, page 403.
In the linker configuration file, an expression is a 65-bit value with the range \(-2^{64}\) to \(2^{64}\). 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**

\[ nr \ [nr\text{-}suffix] \]

where \(nr\) is either a decimal number or a hexadecimal number (0x... or 0X...). and where \(nr\text{-}suffix\) 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 \textit{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 \textit{If directive}, page 421.

- **Dividing the linker configuration file into several different files**

  The \textit{include} directive makes it possible to divide the configuration file into several logically distinct files. See \textit{Include directive}, page 421.
If directive

Syntax
if (expr) {
  directives
} else if (expr) {
  directives
} else {
  directives
}

where expr is an expression, see Expressions, page 419.

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 404.

Include directive

Syntax
include filename;

Parameters

filename A string literal where both / and \ can be used as the directory delimiter.

Description
The include 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.
Structural configuration
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 70.

Summary of sections

This table lists the ELF sections and blocks that are used by the IAR build tools:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bss</td>
<td>Holds zero-initialized static and global variables.</td>
</tr>
<tr>
<td>CSTACK</td>
<td>Holds the stack used by programs.</td>
</tr>
<tr>
<td>.cstart</td>
<td>Holds the startup code.</td>
</tr>
<tr>
<td>.data</td>
<td>Holds static and global initialized variables.</td>
</tr>
<tr>
<td>.data_init</td>
<td>Holds initial values for .data sections when the linker directive initialize by copy is used.</td>
</tr>
<tr>
<td>__DLIB_PERTHREAD</td>
<td>Holds variables that contain static states for DLIB modules.</td>
</tr>
<tr>
<td>.exc.text</td>
<td>Holds exception-related code.</td>
</tr>
<tr>
<td>HEAP</td>
<td>Holds the heap used for dynamically allocated data.</td>
</tr>
<tr>
<td>.iar.dynexit</td>
<td>Holds the atexit table.</td>
</tr>
<tr>
<td>.init_array</td>
<td>Holds a table of dynamic initialization functions.</td>
</tr>
<tr>
<td>.intvec</td>
<td>Holds the reset vector table.</td>
</tr>
<tr>
<td>IRQ_STACK</td>
<td>Holds the stack for interrupt requests, IRQ, and exceptions.</td>
</tr>
<tr>
<td>.noinit</td>
<td>Holds __no_init static and global variables.</td>
</tr>
<tr>
<td>.preinit_array</td>
<td>Holds a table of dynamic initialization functions.</td>
</tr>
<tr>
<td>.prepreinit_array</td>
<td>Holds a table of dynamic initialization functions.</td>
</tr>
<tr>
<td>.rodata</td>
<td>Holds constant data.</td>
</tr>
</tbody>
</table>

Table 42: Section summary
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 73.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.text</code></td>
<td>Holds the program code.</td>
</tr>
<tr>
<td><code>.textrw</code></td>
<td>Holds <code>__ramfunc</code> declared program code.</td>
</tr>
<tr>
<td><code>.textrw_init</code></td>
<td>Holds initializers for the <code>.textrw</code> declared section.</td>
</tr>
</tbody>
</table>

---

### .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

Setting up the stack, page 89.

.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 initialize by copy is used, a corresponding .data_init section is created for each .data 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 .data sections. This section is created by the linker if the initialize by copy linker directive is used.

Memory placement

This section can be placed anywhere in memory.

.DLIB_PERTHREAD

Description

Holds thread-local static and global initialized variables used by the main thread.

This section is placed automatically. If you change the placement, you must not change its initialization. The initialization of this section must be controlled with the initialize by copy directive.

Memory placement

This section can be placed anywhere in memory.

See also

Managing a multithreaded environment, page 130.
<table>
<thead>
<tr>
<th>Section Type</th>
<th>Description</th>
<th>Memory Placement</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td>.exc.text</td>
<td>Holds code that is only executed when your application handles an exception.</td>
<td>In the same memory as .text.</td>
<td><em>Exception handling</em>, page 174.</td>
</tr>
<tr>
<td>HEAP</td>
<td>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>.</td>
<td>This section can be placed anywhere in memory.</td>
<td><em>Setting up the heap</em>, page 90.</td>
</tr>
<tr>
<td>.iar.dynexit</td>
<td>Holds the table of calls to be made at exit.</td>
<td>This section can be placed anywhere in memory.</td>
<td><em>Setting up the atexit limit</em>, page 90.</td>
</tr>
<tr>
<td>.init_array</td>
<td>Holds pointers to routines to call for initializing one or more C++ objects with static storage duration.</td>
<td>This section can be placed anywhere in memory.</td>
<td></td>
</tr>
<tr>
<td>.intvec</td>
<td>Holds the reset vector table and exception vectors which contain branch instructions to <code>cstartup</code>, interrupt service routines etc.</td>
<td>Must be placed at address range 0x00 to 0x3F.</td>
<td></td>
</tr>
</tbody>
</table>
**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 cstartup.s 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**
*Exception stack*, page 184.

**.noinit**

**Description**
Holds static and global __no_init variables.

**Memory placement**
This section can be placed anywhere in memory.

**.preinit_array**

**Description**
Like .init_array, 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**
*.init_array*, page 426.

**.prepreinit_array**

**Description**
Like .init_array, 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**
*.init_array*, page 426.

**.rodata**

**Description**
Holds constant data. This can include constant variables, string and aggregate literals, etc.
Descriptions of sections and blocks

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 __ramfunc declared program code.
Memory placement  This section can be placed anywhere in memory.
See also  __ramfunc, page 321.

.textrw_init
Description  Holds initializers for the .textrw declared sections.
Memory placement  This section can be placed anywhere in memory.
See also  __ramfunc, page 321.
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 78.

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".

C++ NAMES

When you specify the name of a C++ function in a stack usage control file, you must use the name exactly as used by the linker. Both the number and names of parameters, as well as the names of types must match. However, most non-significant white-space differences are accepted. In particular, you must enclose the name in quote marks because all C++ function names include non-identifier characters.

You can also use wildcards in function names. "#*" matches any sequence of characters, and "#?" matches a single character. This makes it possible to write function names that will match any instantiation of a template function.

Examples:

'operator new(unsigned int)*'
'cout::flush()'

std::ostream::flush()'

'operator <<(ostream &, char const *)'

'void _Sort<#*>(#*, #*, #*)'

Stack usage control directives

This section gives detailed reference information about each stack usage control directive.
function directive

Syntax

```plaintext
[ override ] function [ category ] function-spec : stack-size
[ , call-info... ];
```

Parameters

- `category`: See `category`, page 433
- `function-spec`: See `function-spec`, page 433
- `call-info`: See `call-info`, page 434
- `stack-size`: See `stack-size`, page 434

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.

Example

```plaintext
function MyFunc1: 32,
calls MyFunc2,
calls MyFunc3, MyFunc4: 16;
function [interrupt] nmi: 44
```

exclude directive

Syntax

```
exclude function-spec [, function-spec... ];
```

Parameters

- `function-spec`: See `function-spec`, page 433

Description

Excludes the specified functions, and call trees originating with them, from stack usage calculations.

Example

```
exclude fun1, fun2;
```
possible calls directive

Syntax
possible calls calling-func : called-func [, called-func ... ];

Parameters
calling-func See function-spec, page 433
called-func See function-spec, page 433

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 #pragma calls directive if the information about which functions that might be called is available when compiling.

Example
possible calls afun: bfun, cfun;

See also
calls, page 328.

call graph root directive

Syntax
call graph root [category ] : function-spec [, function-spec ... ];

Parameters
category See category, page 433
function-spec See function-spec, page 433

Description
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 Stack Usage chapter in the linker map file.

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.

Example
call graph root [task]: fun1, fun2;

See also
call_graph_root, page 328.
**max recursion depth directive**

**Syntax**

```
max recursion depth function-spec : size;
```

**Parameters**

- `function-spec` See `function-spec`, page 433
- `size` See `size`, page 435

**Description**

Specifies the maximum number of iterations through any of the cycles in the recursion nest of which the function is a member.

A recursion nest is a set of cycles in the call graph where each cycle shares at least one node with another cycle in the nest.

Stack usage analysis will base its result on the max recursion depth multiplied by the stack usage of the deepest cycle in the nest. If the nest is not entered on a point along one of the deepest cycles, no stack usage result will be calculated for such calls.

**Example**

```
max recursion depth fun1: 10;
``` 

**no calls from directive**

**Syntax**

```
no calls from module-spec to function-spec [, function-spec... ];
```

**Parameters**

- `function-spec` See `function-spec`, page 433
- `module-spec` See `module-spec`, page 433

**Description**

When you provide stack usage information for some functions in a module without stack usage information, the linker warns about functions that are referenced from the module but not listed as called. This is primarily to help avoid problems with C runtime routines, calls to which are generated by the compiler, beyond user control.

If there actually is no call to some of these functions, use the `no calls from` directive to selectively suppress the warning for the specified functions. You can also disable the warning entirely (`--diag_suppress` or `Project>Options>Linker>Diagnostics>Suppress these diagnostics`).

**Example**

```
no calls from [file.o] to fun1, fun2;
```
Syntactic components

The stack usage control directives use some syntactical components. These are described below.

**category**

Syntax

\[
\text{[ name ]}
\]

Description

A call graph root category. You can use any name you like. Categories are not case-sensitive.

Example

category examples:

\[
\text{[interrupt]}
\]
\[
\text{[task]}
\]

**function-spec**

Syntax

\[
\text{[ ? ] name [ module-spec ]}
\]

Description

Specifies the name of a symbol, and for module-local symbols, the name of the module it is defined in. Normally, if the function-spec does not match a symbol in the program, a warning is emitted. Prefixing with ? suppresses this warning.

Example

function-spec examples:

\[
xFun
\]
\[
MyFun [file.o]
\]
\[
?\text{fun1(int)}
\]

**module-spec**

Syntax

\[
\text{[name [ (name) ]]}
\]

Description

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:

- The complete path of the file ("D:\CL\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").
Syntactic components

Note that when using multi-file compilation (--mfc), multiple files are compiled into a single module, named after the first file.

Example

module-spec examples:
[file.o]
[file.o(lib.a)]
['D:\C1\test\file.o']

name

Description
A name can be either an identifier or a quoted string.

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.

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 ".

Example

name examples:
MyFun
file.o
"file-1.o"

call-info

Syntax

calls function-spec [, function-spec...][ : stack-size ]

Description
Specifies one or more called functions, and optionally, the stack size at the calls.

Example
call-info examples:
calls MyFunc1 : stack 16
calls MyFunc2, MyFunc3, MyFunc4

stack-size

Syntax

[ stack ] size

Description
Specifies the size of a stack frame.
Example

stack-size examples:
24
stack 28

size

Description
A decimal integer, or 0x followed by a hexadecimal integer. Either alternative can optionally be followed by a suffix indicating a power of two (K=2^{10}, M=2^{20}, G=2^{30}, T=2^{40}, P=2^{50}).

Example

size examples:
24
0x18
2048
2K
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>Command line options that define an operation to be performed. Such an option must be specified before the name of the library file.</td>
</tr>
<tr>
<td>libraryfile</td>
<td>The library file to be operated on.</td>
</tr>
<tr>
<td>objectfile1 ... objectfileN</td>
<td>The object file(s) that the specified command operates on.</td>
</tr>
<tr>
<td>options</td>
<td>Command line options that define actions to be performed. These options can be placed anywhere on the command line.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--create</td>
<td>Creates a library that contains the listed object files.</td>
</tr>
<tr>
<td>--delete, -d</td>
<td>Deletes the listed object files from the library.</td>
</tr>
<tr>
<td>--extract, -x</td>
<td>Extracts the listed object files from the library.</td>
</tr>
<tr>
<td>--replace, -r</td>
<td>Replaces or appends the listed object files to the library.</td>
</tr>
<tr>
<td>--symbols</td>
<td>Lists all symbols defined by files in the library.</td>
</tr>
<tr>
<td>--toc, -t</td>
<td>Lists all files in the library.</td>
</tr>
</tbody>
</table>

Table 44: iarchive commands summary

For more information, see Descriptions of options, page 451.
SUMMARY OF IARCHIVE OPTIONS

This table summarizes the `iarchive` options:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-f</code></td>
<td>Extends the command line.</td>
</tr>
<tr>
<td><code>--output, -o</code></td>
<td>Specifies the library file.</td>
</tr>
<tr>
<td><code>--silent</code></td>
<td>Sets silent operation.</td>
</tr>
<tr>
<td><code>--verbose, -V</code></td>
<td>Reports all performed operations.</td>
</tr>
</tbody>
</table>

For more information, see Descriptions of options, page 451.

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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inputfile</code></td>
<td>An absolute ELF executable image produced by the ILINK linker.</td>
</tr>
<tr>
<td><code>options</code></td>
<td>Any of the available command line options, see <code>Summary of ielftool options</code>, page 441.</td>
</tr>
<tr>
<td><code>outputfile</code></td>
<td>An absolute ELF executable image.</td>
</tr>
</tbody>
</table>

*Table 46: ielftool parameters*

For information about specifying a filename or a directory, see `Rules for specifying a filename or directory as parameters`, page 228.

**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:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--bin</td>
<td>Sets the format of the output file to binary.</td>
</tr>
<tr>
<td>--checksum</td>
<td>Generates a checksum.</td>
</tr>
<tr>
<td>--fill</td>
<td>Specifies fill requirements.</td>
</tr>
<tr>
<td>--ihex</td>
<td>Sets the format of the output file to linear Intel hex.</td>
</tr>
<tr>
<td>--self_reloc</td>
<td>Not for general use.</td>
</tr>
<tr>
<td>--silent</td>
<td>Sets silent operation.</td>
</tr>
<tr>
<td>--simple</td>
<td>Sets the format of the output file to Simple code.</td>
</tr>
<tr>
<td>--srec</td>
<td>Sets the format of the output file to Motorola S-records.</td>
</tr>
</tbody>
</table>

*Table 47: ielftool options summary*
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_file</td>
<td>An ELF relocatable or executable file to use as input.</td>
</tr>
<tr>
<td>output_file</td>
<td>A file or directory where the output is emitted. If absent and no --output option is specified, output is directed to the console.</td>
</tr>
</tbody>
</table>

Table 48: ielfdumparm parameters
For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 228.

**SUMMARY OF IELFDUMPARM OPTIONS**

This table summarizes the `ielfdumparm` command line options:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--all</td>
<td>Generates output for all input sections regardless of their names or numbers.</td>
</tr>
<tr>
<td>--code</td>
<td>Dumps all sections that contain executable code.</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line.</td>
</tr>
<tr>
<td>--output, -o</td>
<td>Specifies an output file.</td>
</tr>
<tr>
<td>--no_strtab</td>
<td>Suppresses dumping of string table sections.</td>
</tr>
<tr>
<td>--raw</td>
<td>Uses the generic hexadecimal/ASCII output format for the contents of any selected section, instead of any dedicated output format for that section.</td>
</tr>
<tr>
<td>--section, -s</td>
<td>Generates output for selected input sections.</td>
</tr>
</tbody>
</table>

| Table 49: ielfdumparm options summary |

For more information, see *Descriptions of options*, page 451.

**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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>options</td>
<td>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.</td>
</tr>
<tr>
<td>inputfile</td>
<td>A relocatable ELF object file.</td>
</tr>
</tbody>
</table>

| Table 50: iobjmanip parameters |
The IAR ELF Object Tool—iobjmanip

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:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f</td>
<td>Extends the command line.</td>
</tr>
<tr>
<td>--remove_section</td>
<td>Removes a section.</td>
</tr>
<tr>
<td>--rename_section</td>
<td>Renames a section.</td>
</tr>
<tr>
<td>--rename_symbol</td>
<td>Renames a symbol.</td>
</tr>
<tr>
<td>--strip</td>
<td>Removes debug information.</td>
</tr>
</tbody>
</table>

For more information, see Descriptions of options, page 241.

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.
The IAR Absolute Symbol Exporter—isymexport

The IAR Absolute Symbol Exporter, isymexport, 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:

```
isymexport [options] inputfile outputfile [options]
```

**Parameters**

The parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inputfile</code></td>
<td>A ROM image in the form of an executable ELF file (output from linking).</td>
</tr>
<tr>
<td><code>options</code></td>
<td>Any of the available command line options, see Summary of isymexport options, page 447.</td>
</tr>
<tr>
<td><code>outputfile</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 52: ielftool parameters

**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`. 
SUMMARY OF ISYMEXPORT OPTIONS

This table summarizes the isymexport command line options:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--edit</td>
<td>Specifies a steering file.</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line.</td>
</tr>
<tr>
<td>--ram_reserve_ranges</td>
<td>Generates symbols to reserve the areas in RAM that the image uses.</td>
</tr>
<tr>
<td>--reserve_ranges</td>
<td>Generates symbols to reserve the areas in ROM and RAM that the image uses.</td>
</tr>
</tbody>
</table>

Table 53: isymexport options summary

For more information, see Descriptions of options, page 451.

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 YYY_* /*Change symbol prefix from xxx_ to YYY_ */
show YYY_* /* Export all symbols from YYY package */
hide *_internal /* But do not export internal symbols */
show zzz? /* Export zzz, but not zzzaaa */
hide zzzx /* But do not export zzz */
Show directive

Syntax
show pattern

Parameters
pattern 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 hide directive.

Example
/* Include all public symbols ending in _pub. */
show *_pub

Hide directive

Syntax
hide pattern

Parameters
pattern 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 show directive.

Example
/* Do not include public symbols ending in _sys. */
hide *_sys

Rename directive

Syntax
rename pattern1 pattern2

Parameters
pattern1 A pattern used for finding symbols to be renamed. The pattern can contain no more than one * or ? wildcard character.

pattern2 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 pattern1.

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 rename 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 isymexport:

Es001: could not open file filename
isymexport 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  --checksum

{symbol[+offset]|address}:size,algorithm[:{1|2}[m][L|W][r][i|p]] [:start];range[:range;...]

Parameters  symbol  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.
offset

An offset to the symbol.

address

The absolute address where the checksum value should be stored.

size

The number of bytes in the checksum: 1, 2, or 4; must not be larger than the size of the checksum symbol.

algorithm

The checksum algorithm used, one of:

• sum, a byte-wise calculated arithmetic sum. The result is truncated to 8 bits.
• sum8wide, a byte-wise calculated arithmetic sum. The result is truncated to the size of the symbol.
• sum32, a word-wise (32 bits) calculated arithmetic sum
• crc16, CRC16 (generating polynomial 0x11021); used by default
• crc32, CRC32 (generating polynomial 0x104C11DB7)
• crc=n, CRC with a generating polynomial of n.

If specified, can be one of:

• 1 – Specifies one’s complement.
• 2 – Specifies two’s complement.

m

Reverses the order of the bits within each byte when calculating the checksum.

L|W

Specifies the size of the unit for which a checksum should be calculated. Typically, this is useful to make the linker produce the same checksum as some hardware CRC implementations that calculate a checksum for more than 8 bits per iteration. Choose between:

L, calculates a checksum on 32 bits in every iteration
W, calculates a checksum on 16 bits in every iteration.

If you do not specify a unit size, 8 bits will be used by default. Using these parameters does not add any additional error detection power to the checksum.

r

Reverses the byte order of the input data within each word of size size.
To calculate a checksum with the specified algorithm for the specified ranges, use the `--checksum` option with the `ielftool` command. The checksum will replace the original value in the symbol. A new absolute symbol will be generated, with the symbol name suffixed with `_value` 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 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 187

To set related options, choose:

- **Project>Options>Linker>Checksum**
Descriptions of options

--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 228.
objectfile1 ... The object file(s) to build the library from.
objectfileN
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, --create is used by default.
This option is not available in the IDE.
--delete, -d

Syntax

```
--delete libraryfile objectfile1 ... objectfileN
-d 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 228.
- **objectfile1 ... objectfileN**: The object file(s) that the command operates on.

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

```
--edit steering_file
```

Tool

`isymexport`

Description

Use this option to specify a steering file to control which symbols that are included in the `isymexport` output file, and also to rename some of the symbols if that is desired.

See also

*Steering files*, page 447.

This option is not available in the IDE.
## --extract, -x

**Syntax**

```
--extract libraryfile [objectfile1 ... objectfileN]
-x 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 228.
- `objectfile1 ... objectfileN`: The object file(s) that the command operates on.

**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**

```
-f filename
```

**Parameters**

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 228.

**Tool**

`iarchive, ielfdumparm, iobjmanip, and isymexport`

**Description**

Use this option to make the tool 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 can 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.

This option is not available in the IDE.
--fill

Syntax

--fill pattern;range[;range...]

Parameters

range

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.

pattern

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 --fill 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

--ihex

Tool

ielftool

Description

Sets the format of the output file to linear Intel hex.

To set related options, choose:

Project>Options>Linker>Output converter
Descriptions of options

--no_strtab
Syntax
--no_strtab
Tool
ielfdumparm
Description
Use this option to suppress dumping of string table sections (sections of type SHT_STRTAB).
This option is not available in the IDE.

--output, -o
Syntax
-o {filename|directory}
--output {filename|directory}
Parameters
For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 228.
Tool
iarchive and ielfdumparm.
Description
iarchive
By default, iarchive assumes that the first argument after the iarchive command is the name of the destination library. Use this option to explicitly specify a different filename for the library.
ielfdumparm
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 id filename extension
You can also specify the output file by specifying a file or directory following the name of the input file.
This option is not available in the IDE.
---ram_reserve_ranges

Syntax:  
```
--ram_reserve_ranges[=symbol_prefix]
```

Parameters:

- `symbol_prefix`: The prefix of symbols created by this option.

Tool:  

- `isymexport`

Description:  

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 `symbol_prefix`.

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.

If `--ram_reserve_ranges` is used together with `--reserve_ranges`, the RAM areas will get their prefix from the `--ram_reserve_ranges` option and the non-RAM areas will get their prefix from the `--reserve_ranges` option.

See also:  

- `--reserve_ranges`, page 462.

This option is not available in the IDE.

---raw

Syntax:  

```
--raw
```

Tool:  

- `ielfdumparm`

Description:  

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.

The generic text format dumps each byte in the section in hexadecimal format, and where appropriate, as ASCII text.

This option is not available in the IDE.
Descriptions of options

--remove_section

Syntax

--remove_section {section|number}

Parameters

section The section—or sections, if there are more than one section with the same name—to be removed.

number The number of the section to be removed. Section numbers can be obtained from an object dump created using ielfdumparm.

Tool iobjmanip

Description Use this option to make iobjmanip omit the specified section when generating the output file.

This option is not available in the IDE.

--rename_section

Syntax

--rename_section {oldname|oldnumber}=newname

Parameters

oldname The section—or sections, if there are more than one section with the same name—to be renamed.

oldnumber The number of the section to be renamed. Section numbers can be obtained from an object dump created using ielfdumparm.

newname The new name of the section.

Tool iobjmanip

Description Use this option to make iobjmanip rename the specified section when generating the output file.

This option is not available in the IDE.
--rename_symbol

Syntax

`--rename_symbol oldname =newname`

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldname</td>
<td>The symbol to be renamed.</td>
</tr>
<tr>
<td>newname</td>
<td>The new name of the symbol.</td>
</tr>
</tbody>
</table>

Tool

iobjmanip

Description

Use this option to make `iobjmanip` rename the specified symbol when generating the output file.

This option is not available in the IDE.

--replace, -r

Syntax

`--replace libraryfile objectfile1 ... objectfileN`

`-r libraryfile objectfile1 ... objectfileN`

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraryfile</td>
<td>The library file that the command operates on.</td>
</tr>
<tr>
<td>objectfile1</td>
<td>The object file(s) that the command operates on.</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>objectfileN</td>
<td></td>
</tr>
</tbody>
</table>

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.
Descriptions of options

--reserve_ranges

Syntax  --reserve_ranges[=symbol_prefix]

Parameters

symbol_prefix  The prefix of symbols created by this option.

Tool  isymexport

Description  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 symbol_prefix.

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.

If --reserve_ranges is used together with --ram_reserve_ranges, the RAM areas will get their prefix from the --ram_reserve_ranges option and the non-RAM areas will get their prefix from the --reserve_ranges option.

See also  --ram_reserve_ranges, page 459.

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**

Causes the tool to operate without sending any messages to the standard output stream.

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.

This option is not available in the IDE.
### --simple

<table>
<thead>
<tr>
<th>Syntax</th>
<th>--simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>ielftool</td>
</tr>
<tr>
<td>Description</td>
<td>Sets the format of the output file to Simple code.</td>
</tr>
</tbody>
</table>

To set related options, choose:

`Project>Options>Output converter`

### --srec

<table>
<thead>
<tr>
<th>Syntax</th>
<th>--srec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>ielftool</td>
</tr>
<tr>
<td>Description</td>
<td>Sets the format of the output file to Motorola S-records.</td>
</tr>
</tbody>
</table>

To set related options, choose:

`Project>Options>Output converter`

### --srec-len

<table>
<thead>
<tr>
<th>Syntax</th>
<th>--srec-len=length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>length (\text{The number of data bytes in each S-record.})</td>
</tr>
<tr>
<td>Tool</td>
<td>ielftool</td>
</tr>
<tr>
<td>Description</td>
<td>Restricts the number of data bytes in each S-record. This option can be used in combination with the --srec option.</td>
</tr>
</tbody>
</table>

This option is not available in the IDE.
--srec-s3only

Syntax
--srec-s3only

Tool
ielftool

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 --srec option.

This option is not available in the IDE.

--strip

Syntax
--strip

Tool
iobjmanip and ielftool.

Description
Use this option to remove all sections containing debug information before the output file is written.

Note that ielftool needs an unstripped input ELF image. If you use the --strip option in the linker, remove it and use the --strip option in ielftool instead.

To set related options, choose:
Project>Options>Linker>Output>Include debug information in output

--symbols

Syntax
--symbols libraryfile

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 228.

Tool
iarchive

Description
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.
Descriptions of options

In silent mode (--silent), this command performs symbol table-related syntax checks on the library file and displays only errors and warnings.

This option is not available in the IDE.

--toc, -t

Syntax
--toc libraryfile
-t libraryfile

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 228.

Tool iarchive

Description Use this command to list the names of all object files (modules) in a specified library. In silent mode (--silent), this command performs basic syntax checks on the library file, and displays only errors and warnings.

This option is not available in the IDE.

--verbose, -V

Syntax
--verbose
-V (iarchive only)

Tool iarchive and ielftool.

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 483. For a short overview of the differences between Standard C and C89, see C language overview, page 161.

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:

```c
int main(void)
```
To change this behavior, see Customizing system initialization, page 117.

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 123.

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 298.

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 306.

ptrdiff_t (6.5.6)
For information about ptrdiff_t, see ptrdiff_t, page 306.

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 Inlining functions, page 65.
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 300.

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 242.

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 300.

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 297.

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 309.

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 381.
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 235.

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 221.

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 221.

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
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 99.

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 395.

Feraiseexcept raising floating-point exception (7.6.2.3)
For information about the feraiseexcept function raising floating-point exceptions,
see Floating-point environment, page 304.

Strings passed to the setlocale function (7.11.1.1)
For information about strings passed to the setlocale function, see Locale, page 123.

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 127.

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 123.

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 123.

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 123.

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 127.

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 127.

clock() (7.23.2.1)
The application must supply an implementation of the clock function. See Time, page 127.

%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 127.

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 297.

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 llong.

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 297.
The value of the result of the sizeof operator (6.5.3.4)
See Data representation, page 297.

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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EZERO</td>
<td>no error</td>
</tr>
<tr>
<td>EDOM</td>
<td>domain error</td>
</tr>
<tr>
<td>ERANGE</td>
<td>range error</td>
</tr>
<tr>
<td>EFPOS</td>
<td>file positioning error</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>multi-byte encoding error</td>
</tr>
<tr>
<td>&lt;0</td>
<td></td>
</tr>
<tr>
<td>all others</td>
<td>error nnn</td>
</tr>
</tbody>
</table>

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 467. For a short overview of the differences between Standard C and C89, see C language overview, page 161.

Descriptions of implementation-defined behavior

The descriptions follow 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 117.
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 123.

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 123.

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 298, 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 304, 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 306, for information about size_t.

Conversion from/to pointers (6.3.4)
See Casting, page 306, for information about casting of data pointers and function pointers.
ptrdiff_t (6.3.6, 7.1.1)
See pptrdiff_t, page 306, for information about the pptrdiff_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 298, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)
A 'plain' int bitfield is treated as an 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
dataset
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_default_requirements
library_provides
library_requirement_override
memory
module_name
no_pch
once
Descriptions of implementation-defined behavior

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 127.

**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 123.

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 123.
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 123.

%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 126.

system() (7.10.4.5)
How the command processor works depends on how you have implemented the system function. See Environment interaction, page 126.

Message returned by strerror() (7.11.6.2)
The messages returned by strerror() depending on the argument is:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EZERO</td>
<td>no error</td>
</tr>
<tr>
<td>EDOM</td>
<td>domain error</td>
</tr>
<tr>
<td>ERANGE</td>
<td>range error</td>
</tr>
<tr>
<td>EFPOS</td>
<td>file positioning error</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>multi-byte encoding error</td>
</tr>
<tr>
<td>&lt;0</td>
<td>&gt;99</td>
</tr>
<tr>
<td>all others</td>
<td>error nnn</td>
</tr>
</tbody>
</table>

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 127.

clock() (7.12.2.1)
From where the system clock starts counting depends on how you have implemented the clock function. See Time, page 127.
Descriptions of implementation-defined behavior
A

__AAPCS__ (predefined symbol) .......................... 382
--aapcs (compiler option) .............................. 233
__AAPCS_VFP__ (predefined symbol) ...................... 382
ABI, AEABI and IA64 .................................. 192
abort
implementation-defined behavior ......................... 479
implementation-defined behavior in C89 (DLIB) .... 492
system termination (DLIB) .............................. 116
__absolute (extended keyword) ......................... 317
absolute location
data, placing at (@) .................................. 202
language support for .................................. 164
placing data in registers (@) ........................... 204
#pragma location ..................................... 333
--aeabi (compiler option) ..................... 234
__AEABI_PORTABILITY_LEVEL (preprocessor symbol) . 194
__AEABI_PORTABLE (preprocessor symbol) .............. 194
algorithm (STL header file) ........................... 393
alignment ............................................ 297
forcing stricter (#pragma data_alignment) ........... 329
in structures (#pragma pack) .......................... 337
in structures, causing problems ......................... 199
of an object (__ALIGNOF__) ............................ 164
of data types ....................................... 297
restrictions for inline assembler ...................... 139
alignment (pragma directive) ......................... 474, 489
__ALIGNOF__ (operator) ................................ 164
--align_sp_on_irq (compiler option) ................. 234
--all (ielfdump option) ................................ 451
anonymous structures .................................. 199
anonymous symbols, creating .......................... 161
ANSI C. See C89
application
building, overview of ................................ 47
execution, overview of ................................ 42
startup and termination (DLIB) ......................... 113
architecture
more information about ................................. 21
of ARM ........................................... 468
argy (argument), implementation-defined behavior ...... 53
ARM
and Thumb code, overview .............................. 57
memory layout ....................................... 53
supported devices ..................................... 34
__arm (extended keyword) .............................. 317
--arm (compiler option) .............................. 234
__ARMVFP__ (predefined symbol) ....................... 383
__ARMVFPV2__ (predefined symbol) ..................... 383
__ARMVFPV3__ (predefined symbol) ..................... 383
__ARMVFPV4__ (predefined symbol) ..................... 383
__ARMVFP_D16__ (predefined symbol) ................... 383
__ARMVFP_FP16__ (predefined symbol) ................. 383
__ARMVFP_SP__ (predefined symbol) .................... 383
__ARMVFP_ALL__ (predefined symbol) ................. 383
__ARMVFP_SP__ (predefined symbol) .................... 383
__ARM_ADVANCED_SIMD__ (predefined symbol) ...... 382
__ARM Media__ (predefined symbol) .................... 382
__ARM_PROFILE_M__ (predefined symbol) ............. 383
__ARM4TM__ (predefined symbol) ....................... 384
__ARM5__ (predefined symbol) ......................... 384
__ARM5E__ (predefined symbol) ......................... 384
__ARM6__ (predefined symbol) ......................... 384
__ARM6M__ (predefined symbol) ......................... 384
__ARM6SM__ (predefined symbol) ....................... 384
__ARM7A__ (predefined symbol) ......................... 384
__ARM7EM__ (predefined symbol) ....................... 384
__ARM7M__ (predefined symbol) ......................... 384
__ARM7R__ (predefined symbol) ......................... 384
arrays
designated initializers in ............................. 161
implementation-defined behavior ...................... 472
implementation-defined behavior in C89 ............... 486
incomplete at end of structs ......................... 161
non-lvalue ....................................... 167
of incomplete types ................................ 166
single-value initialization ............................ 168
asm, __asm (language extension) ...................... 140
assembler code
  calling from C ............................................. 147
  calling from C++ ........................................... 149
  inserting inline ............................................ 139
assembler directives
  for call frame information ................................. 157
  using in inline assembler code ............................ 139
assembler instructions
  for software interrupts ................................. 62
  inserting inline ............................................ 139
  assembler labels, making public (--public_equ) ......... 261
assembler language interface ................................. 137
  calling convention. See assembler code
assembler list file, generating ................................ 248
assembler output file ......................................... 149
asserts ........................................................... 129
  implementation-defined behavior of ..................... 476
  implementation-defined behavior of in C89, (DLIB) .... 490
  including in application ..................................... 388
assert.h (DLIB header file) ..................................... 391
__assignment_by_bitwise_copy_allowed, symbol used in library .................................................. 397
@ (operator) ...................................................... 310
  placing at absolute address ............................. 202
  placing in sections ........................................ 203
atexit ........................................................... 130
  reserving space for calls .................................. 90
atexit limit, setting up ....................................... 90
attributes
  object ......................................................... 315
  type .......................................................... 313
auto variables .................................................. 54
  at function entrance ....................................... 152
programming hints for efficient code ......................... 211
  using in inline assembler statements ..................... 140
auto, packing algorithm for initializers ....................... 409

backtrace information See call frame information
Barr, Michael .................................................. 25
baseaddr (pragma directive) .................................. 474, 489
__BASE_FILE__ (predefined symbol) ......................... 384
basic_template_matching (pragma directive) .................. 474, 489
batch files
  error return codes ......................................... 223
  none for building library from command line .............. 112
  --BE8 (linker option) ...................................... 272
  --BE32 (linker option) ................................... 272, 275
  --BE8 (linker option) ...................................... 272
  __big_endian (extended keyword) ........................... 317
  big-endian (byte order) .................................... 49
  --bin (ielftool option) .................................... 451
binary streams ................................................. 477
  binary streams in C89 (DLIB) ............................... 491
  bit negation ................................................ 213
bitfields
  data representation of ..................................... 300
  hints ........................................................ 198
  implementation-defined behavior ......................... 473
  implementation-defined behavior in C89 .................. 487
  non-standard types in ...................................... 164
  bitfields (pragma directive) ............................... 327
  bits in a byte, implementation-defined behavior .......... 469
  bitset (library header file) ................................. 393
  bold style, in this guide ................................. 26
  bool (data type) ............................................. 298
    adding support for in DLIB .............................. 392, 395
    .bss (ELF section) ....................................... 424
    building_runtime (pragma directive) .................... 474, 489
    __BUILD_NUMBER__ (predefined symbol) .................. 384
  Burrows-Wheeler algorithm, for packing initializers ...... 409
  bwt, packing algorithm for initializers ................... 409
  byte order .................................................. 49
  identifying ................................................ 386
  specifying (--endian) ....................................... 244
C

C and C++ linkage ............................................ 151
C/C++ calling convention. See calling convention
C header files ................................................ 391
C language, overview ....................................... 161
call frame information ..................................... 157
   in assembler list file ................................... 149
   in assembler list file (-lA) .......................... 248
call graph root (stack usage control directive) .... 431
call stack ..................................................... 157
calleesave registers, stored on stack ................. 54
calling convention
   C++, requiring C linkage ............................... 149
   in compiler ............................................. 150
calloc (library function) ................................ 55
   See also heap
   implementation-defined behavior in C89 (DLIB) ... 492
calls (pragma directive) .................................. 328
   --call_graph (compiler option) .................... 273
call_graph_root (pragma directive) ................... 328
call_info (in stack usage control file) .............. 434
can_instantiate (pragma directive) ................... 474, 489
cassert (library header file) ......................... 394
cast operators
   in Extended EC++ .................................... 172, 179
   missing from Embedded C++ .......................... 172
casting
   of pointers and integers ............................ 306
   pointers to integers, language extension ......... 167
category (in stack usage control file) ............... 433
ccomplex (library header file) ......................... 394
cctype (DLIB header file) .............................. 394
cerrno (DLIB header file) .............................. 394
cexit (system termination code), in DLIB .......... 113
cfenv (library header file) ........................... 394
CFI (assembler directive) .............................. 157
CFI_COMMON_ARM (call frame information macro) ... 160
CFI_COMMON Thumb (call frame information macro) 160
CFI_NAMES_BLOCK (call frame information macro) ... 160
cfloat (DLIB header file) .............................. 394
char (data type) .......................................... 298
   changing default representation (--char_is_unsigned) 235
   changing representation (--char_is_unsigned) ...... 235
   implementation-defined behavior .................. 469
   signed and unsigned ................................ 299
character set, implementation-defined behavior .... 468
characters, implementation-defined behavior ......... 469
characters, implementation-defined behavior in C89 ... 484
character-based I/O, in DLIB ......................... 119
   --char_is_signed (compiler option) .............. 235
   --char_is_unsigned (compiler option) .......... 235
check that (linker directive) .......................... 417
checksum
   calculation of ......................................... 187
   display format in C-SPY for symbol .............. 192
   --checksum (ielftool option) ..................... 451
   cinttypes (DLIB header file) ..................... 394
ciso646 (library header file) ....................... 394
climits (DLIB header file) ........................... 394
clobber .................................................... 140
clocale (DLIB header file) ........................... 394
clock (DLIB library function),
   implementation-defined behavior in C89 .......... 493
clock (library function)
   implementation-defined behavior ................ 480
   clock.c ................................................ 127
   _close (DLIB library function) .................... 123
   _CLREX (intrinsic function) ....................... 352
   clustering (compiler transformation) ............. 210
   disabling (--no_clustering) ....................... 251
   __CLZ (intrinsic function) ......................... 352
   cmem (system initialization code), in DLIB .... 113
cmath (DLIB header file) ............................. 395
CMIS integration .......................................... 195
code
   ARM and Thumb, overview ............................ 57
   interruption of execution .......................... 59
   --code (ielfdump option) .......................... 454
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>code motion (compiler transformation)</td>
<td>209</td>
</tr>
<tr>
<td>disabling (--no_code_motion)</td>
<td>251</td>
</tr>
<tr>
<td>codeseg (pragma directive)</td>
<td>475</td>
</tr>
<tr>
<td>__code, symbol used in library</td>
<td>397</td>
</tr>
<tr>
<td><strong>command line options</strong></td>
<td></td>
</tr>
<tr>
<td><em>See also</em> compiler options</td>
<td></td>
</tr>
<tr>
<td><em>See also</em> linker options</td>
<td></td>
</tr>
<tr>
<td>part of compiler invocation syntax</td>
<td>219</td>
</tr>
<tr>
<td>part of linker invocation syntax</td>
<td>219</td>
</tr>
<tr>
<td>passing</td>
<td>220</td>
</tr>
<tr>
<td>typographic convention</td>
<td>26</td>
</tr>
<tr>
<td>command prompt icon, in this guide</td>
<td>27</td>
</tr>
<tr>
<td>.comment (ELF section)</td>
<td>424</td>
</tr>
<tr>
<td><strong>comments</strong></td>
<td></td>
</tr>
<tr>
<td>after preprocessor directives</td>
<td>167</td>
</tr>
<tr>
<td>C++ style, using in C code</td>
<td>161</td>
</tr>
<tr>
<td>common block (call frame information)</td>
<td>157</td>
</tr>
<tr>
<td>common subexpr elimination (compiler transformation)</td>
<td>208</td>
</tr>
<tr>
<td>disabling (--no_cse)</td>
<td>252</td>
</tr>
<tr>
<td>Common.i (CFI header example file)</td>
<td>160</td>
</tr>
<tr>
<td><strong>compilation date</strong></td>
<td></td>
</tr>
<tr>
<td>exact time of (<strong>TIME</strong>)</td>
<td>387</td>
</tr>
<tr>
<td>identifying (<strong>DATE</strong>)</td>
<td>384</td>
</tr>
<tr>
<td><strong>compiler</strong></td>
<td></td>
</tr>
<tr>
<td>environment variables</td>
<td>220</td>
</tr>
<tr>
<td>invocation syntax</td>
<td>219</td>
</tr>
<tr>
<td>output from</td>
<td>222</td>
</tr>
<tr>
<td>compiler listing. generating (-l)</td>
<td>248</td>
</tr>
<tr>
<td>compiler object file</td>
<td>40</td>
</tr>
<tr>
<td>including debug information in (--debug, -r)</td>
<td>238</td>
</tr>
<tr>
<td>output from compiler</td>
<td>222</td>
</tr>
<tr>
<td><strong>compiler options</strong></td>
<td></td>
</tr>
<tr>
<td>compiler optimization levels</td>
<td>207</td>
</tr>
<tr>
<td>compiler options</td>
<td>227</td>
</tr>
<tr>
<td>passing to compiler</td>
<td>220</td>
</tr>
<tr>
<td>reading from file (-f)</td>
<td>245</td>
</tr>
<tr>
<td>specifying parameters</td>
<td>229</td>
</tr>
<tr>
<td>summary</td>
<td>229</td>
</tr>
<tr>
<td>syntax</td>
<td>227</td>
</tr>
<tr>
<td>for creating skeleton code</td>
<td>148</td>
</tr>
<tr>
<td>instruction scheduling</td>
<td>210</td>
</tr>
<tr>
<td>--warnings_affect_exit_code</td>
<td>223</td>
</tr>
<tr>
<td>compiler platform, identifying</td>
<td>386</td>
</tr>
<tr>
<td>compiler transformations</td>
<td>205</td>
</tr>
<tr>
<td>compiler version number</td>
<td>387</td>
</tr>
<tr>
<td>compiling</td>
<td></td>
</tr>
<tr>
<td>from the command line</td>
<td>47</td>
</tr>
<tr>
<td>syntax</td>
<td>219</td>
</tr>
<tr>
<td>complex numbers, supported in Embedded C++</td>
<td>172</td>
</tr>
<tr>
<td>complex (library header file)</td>
<td>393</td>
</tr>
<tr>
<td>complex.h (library header file)</td>
<td>391</td>
</tr>
<tr>
<td>compound literals</td>
<td>161</td>
</tr>
<tr>
<td>computer style, typographic convention</td>
<td>26</td>
</tr>
<tr>
<td>--config (linker option)</td>
<td>273</td>
</tr>
<tr>
<td><strong>configuration</strong></td>
<td></td>
</tr>
<tr>
<td>basic project settings</td>
<td>48</td>
</tr>
<tr>
<td>__low_level_init</td>
<td>117</td>
</tr>
<tr>
<td><strong>configuration file for linker</strong></td>
<td></td>
</tr>
<tr>
<td>See linker configuration file</td>
<td></td>
</tr>
<tr>
<td><strong>configuration symbols</strong></td>
<td></td>
</tr>
<tr>
<td>for file input and output</td>
<td>123</td>
</tr>
<tr>
<td>for locale</td>
<td>124</td>
</tr>
<tr>
<td>for printf and scanf</td>
<td>121</td>
</tr>
<tr>
<td>in library configuration files</td>
<td>113, 118</td>
</tr>
<tr>
<td>in linker configuration files</td>
<td>418</td>
</tr>
<tr>
<td>specifying for linker</td>
<td>274</td>
</tr>
<tr>
<td>--config_def (linker option)</td>
<td>274</td>
</tr>
<tr>
<td>consistency, module</td>
<td>135</td>
</tr>
<tr>
<td><strong>const</strong></td>
<td></td>
</tr>
<tr>
<td>declaring objects</td>
<td>310</td>
</tr>
<tr>
<td>non-top level</td>
<td>167</td>
</tr>
<tr>
<td>__constrange(), symbol used in library</td>
<td>397</td>
</tr>
<tr>
<td>__construction_by_bitwise_copy_allowed, symbol used in library</td>
<td>397</td>
</tr>
<tr>
<td>constseg (pragma directive)</td>
<td>475, 489</td>
</tr>
<tr>
<td>const_cast (cast operator)</td>
<td>172</td>
</tr>
<tr>
<td>contents, of this guide</td>
<td>21</td>
</tr>
<tr>
<td>control characters,</td>
<td></td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>481</td>
</tr>
<tr>
<td>conventions, used in this guide</td>
<td>26</td>
</tr>
<tr>
<td><strong>copyright notice</strong></td>
<td>2</td>
</tr>
</tbody>
</table>
data pointers ........................................... 306
data types ............................................ 298
floating point ......................................... 304
in C++ .................................................. 311
integer types ........................................... 298
dataseg (pragma directive) ......................... 475, 489
data_alignment (pragma directive) ............... 329
.data_init (ELF section) ............................. 425
__data, symbol used in library .................... 397
__DATE__ (predefined symbol) .................... 384
date (library function), configuring support for .. 127
DC32 (assembler directive) ......................... 139
--debug (compiler option) ......................... 238
--diag_warning (linker option) .................. 277
--diag_warning (compiler option) ............... 240
diag_suppress (pragma directive) ............... 330
--diag_suppress (linker option) ................. 277
--diag_suppress (compiler option) .............. 240
diag_remark (pragma directive) ................. 330
--diag_remark (linker option) .................. 276
--diag_remark (compiler option) ............... 240
diagnostic messages ................................. 224
classifying as compilation errors ............... 239
classifying as compilation remarks ............. 240
classifying as compiler warnings ............... 240
classifying as errors ............................... 252, 287
classifying as linker warnings .................. 277
classifying as linking errors ..................... 276
classifying as linking remarks ................... 276
disabling compiler warnings ...................... 258
disabling linker warnings ......................... 289
disabling wrapping of in compiler ............... 258
disabling wrapping of in compiler ............... 290
--no_fragments (linker option) ................. 287
--no_fragments (compiler option) .............. 252
listing all used by compiler ..................... 241
listing all used by linker ....................... 278
suppressing in compiler ......................... 240
suppressing in linker .............................. 277
diagnostics
--destroy (linker option) ......................... 287
--destroy (compiler option) ..................... 252
diagnostic messages ................................. 224
classifying as compile time errors .............. 239
classifying as compilation remarks ............. 240
classifying as compiler warnings ............... 240
classifying as errors ............................... 252, 287
classifying as linker warnings .................. 277
classifying as linking errors ..................... 276
classifying as linking remarks ................... 276
disabling compiler warnings ...................... 258
disabling linker warnings ......................... 289
disabling wrapping of in compiler ............... 258
disabling wrapping of in compiler ............... 290
--no_fragments (linker option) ................. 258
--no_fragments (compiler option) .............. 252
listing all used by compiler ..................... 241
listing all used by linker ....................... 278
suppressing in compiler ......................... 240
suppressing in linker .............................. 277
declarations
empty .................................................. 168
in for loops .......................................... 161
Kernighan & Ritchie ................................. 212
of functions .......................................... 151
declarations and statements, mixing ............. 161
declartors, implementation-defined behavior in C89 . 488
define block (linker directive) ................... 405
define memory (linker directive) .................. 400
define overlay (linker directive) ............... 407
define region (linker directive) ................. 401
define symbol (linker directive) ................ 418
--define_symbol (linker option) ................. 275
define_type_info (pragma directive) .......... 475, 489
--delete (iarchive option) ....................... 455
delete (keyword) .................................... 55
denormalized numbers. See subnormal numbers
--dependencies (compiler option) ............... 238
--dependencies (linker option) ................. 275
dequeue (STL header file) ......................... 393
destructors and interrupts, using ................ 177
device description files, preconfigured for C-SPY . 34
<table>
<thead>
<tr>
<th>Directive/Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>diag_warning</td>
<td>330</td>
</tr>
<tr>
<td>directives</td>
<td>330</td>
</tr>
<tr>
<td>pragma</td>
<td>35</td>
</tr>
<tr>
<td>do not instantiate</td>
<td>475</td>
</tr>
<tr>
<td>do_not_instantiate</td>
<td>475</td>
</tr>
<tr>
<td>dynamic memory</td>
<td>55</td>
</tr>
<tr>
<td>dynamic RTTI data, including in the image</td>
<td>286</td>
</tr>
<tr>
<td>__DSB</td>
<td>353</td>
</tr>
<tr>
<td>_disable_irq</td>
<td>353</td>
</tr>
<tr>
<td>_disable_fiq</td>
<td>353</td>
</tr>
<tr>
<td>_disable_irq</td>
<td>353</td>
</tr>
<tr>
<td><strong>DOUBLE</strong></td>
<td>385</td>
</tr>
<tr>
<td>DLIB</td>
<td>490</td>
</tr>
<tr>
<td><strong>DOUBLE</strong></td>
<td>385</td>
</tr>
<tr>
<td>double</td>
<td>304</td>
</tr>
<tr>
<td>_DSB</td>
<td>353</td>
</tr>
<tr>
<td>dynamic initialization</td>
<td>113</td>
</tr>
<tr>
<td>dynamic memory</td>
<td>55</td>
</tr>
<tr>
<td>Diag Warning</td>
<td>330</td>
</tr>
<tr>
<td>DLIB</td>
<td>490</td>
</tr>
<tr>
<td><strong>DOUBLE</strong></td>
<td>385</td>
</tr>
<tr>
<td>double</td>
<td>304</td>
</tr>
<tr>
<td>_DSB</td>
<td>353</td>
</tr>
<tr>
<td>dynamic initialization</td>
<td>113</td>
</tr>
<tr>
<td>dynamic memory</td>
<td>55</td>
</tr>
<tr>
<td>dynamic RTTI data, including in the image</td>
<td>286</td>
</tr>
</tbody>
</table>

---

**E**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e</td>
<td>242</td>
</tr>
<tr>
<td>early_initialization</td>
<td>475</td>
</tr>
<tr>
<td>--ec++</td>
<td>243</td>
</tr>
<tr>
<td>--edit</td>
<td>455</td>
</tr>
<tr>
<td>--enable_multibytes</td>
<td>244</td>
</tr>
<tr>
<td>--endian</td>
<td>244</td>
</tr>
<tr>
<td>--ec++</td>
<td>243</td>
</tr>
<tr>
<td>ELF</td>
<td>437</td>
</tr>
<tr>
<td>Embedded C++</td>
<td>171</td>
</tr>
<tr>
<td>--endian</td>
<td>171</td>
</tr>
<tr>
<td>embedded systems</td>
<td>34</td>
</tr>
<tr>
<td>_embedded_cplusplus</td>
<td>385</td>
</tr>
<tr>
<td>empty region</td>
<td>404</td>
</tr>
<tr>
<td>--enable_irq</td>
<td>354</td>
</tr>
<tr>
<td>--enable_irq</td>
<td>354</td>
</tr>
<tr>
<td>--enable_interrrupt</td>
<td>354</td>
</tr>
<tr>
<td>--enable_irq</td>
<td>354</td>
</tr>
<tr>
<td>--enable_multibytes</td>
<td>244</td>
</tr>
<tr>
<td>--endian</td>
<td>244</td>
</tr>
<tr>
<td>endianness</td>
<td>244</td>
</tr>
<tr>
<td>entry</td>
<td>279</td>
</tr>
<tr>
<td>enums</td>
<td>473</td>
</tr>
<tr>
<td>enumerations</td>
<td>473</td>
</tr>
<tr>
<td>environment</td>
<td>468</td>
</tr>
<tr>
<td>implementation-defined behavior in C89</td>
<td>483</td>
</tr>
<tr>
<td>runtime</td>
<td>99</td>
</tr>
</tbody>
</table>
environment names, implementation-defined behavior . . . 469
environment variables
C_INCLUDE ........................................... 220–221
ILINKARM_CMD_LINE ................................. 221
QCCARM ............................................. 220
environment (native),
implementation-defined behavior ................... 482
EQU (assembler directive) ......................... 261
ERANGE ............................................. 476
ERANGE (C89) ........................................ 490
errno value at underflow,
implementation-defined behavior .................. 479
errno.h (library header file) ....................... 391–392
error messages ........................................ 225
classifying ......................................... 252, 287
classifying for compiler ............................. 239
classifying for linker ............................... 276
range .................................................. 95
error return codes .................................... 223
error (pragma directive) ............................ 331
–error_limit (compiler option) ................... 245
–error_limit (linker option) ...................... 279
escape sequences, implementation-defined behavior . . . 469
exception flags, for floating-point values ........... 304
exception handling, missing from Embedded C++ .... 171
exception (library header file) ..................... 393
exceptions, code for in section ..................... 426
–exception_tables (linker option) ............... 280
exclude (stack usage control directive) ............ 430
.exctext (ELF section) ............................... 426
_exit (library function) ............................. 116
exit (library function) ................................ 116
implementation-defined behavior .................. 479
implementation-defined behavior in C89 ............ 492
_exit (library function) ............................. 116
_exp (library routine) ................................ 116
expf (library routine) ................................. 128
expl (library routine) ................................ 128
export keyword, missing from Extended EC++ ...... 178
export (linker directive) ............................ 418
–export_builtins (linker option) ................. 280
expressions (in linker configuration file) .......... 419
extended command line file
for compiler ........................................... 245
for linker ............................................. 281
passing options ....................................... 220
Extended Embedded C++ ............................ 172
enabling ............................................. 243
extended keywords ................................... 313
enabling (-e) ......................................... 242
overview ............................................. 35
summary ............................................. 315
syntax
object attributes ...................................... 315
type attributes on data objects ..................... 314
type attributes on data pointers .................... 314
type attributes on functions ......................... 314
extended-selectors (in linker configuration file) .... 413, 416
extern "C" linkage .................................... 176
–extract (archive option) ......................... 456
–extra_init (linker option) ....................... 281

F

-f (compiler option) ................................. 245
-f (IAR utility option) ............................. 456
-f (linker option) ................................... 281
fast interrupts ....................................... 61
fatal error messages ................................ 225
fdopen, in stdio.h .................................... 396
fegetraerdisable ..................................... 395
fegettrapenable ..................................... 395
FENV_ACCESS, implementation-defined behavior ...... 472
fenv.h (library header file) ......................... 392, 394
additional C functionality .......................... 395
fgetpos (library function), implementation-defined
behavior ............................................. 479
declaring ............................................. 151, 212
inlining ............................................. 161, 209, 211, 332
interrupt ............................................. 59
intrinsic ............................................. 137, 211
parameters ........................................... 152
placing in memory ................................ 201, 203, 263
recursive
  avoiding ........................................... 211
  storing data on stack ......................... 54
reentrancy (DLIB) .................................. 390
related extensions ................................ 57
return values from ................................ 154
special function types ............................. 59
function effects (pragma directive) .......... 475, 489
function_spec (in stack usage control file) 433

getenv (library function), configuring support for ........ 126
getw, in stdio.h ..................................... 396
getzone (library function), configuring support for ....... 127
getzone.c ............................................. 127
  __get_BASEPRI (intrinsic function) .......... 354
  __get_CONTROL (intrinsic function) ........ 354
  __get_CPSR (intrinsic function) .......... 354
  __get_FAULTMASK (intrinsic function) ..... 355
  __get_FPSCR (intrinsic function) .......... 355
  __get_interrupt_state (intrinsic function) 355
  __get_IPSR (intrinsic function) .......... 356
  __get_LR (intrinsic function) ............. 356
  __get_MSP (intrinsic function) .......... 356
  __get_PRIMASK (intrinsic function) ...... 356
  __get_PSP (intrinsic function) .......... 356
  __get_PSR (intrinsic function) .......... 357
  __get_SB (intrinsic function) .......... 357
  __get_SP (intrinsic function) ............ 357
GRP_COMDAT, group type ......................... 445
--guard_calls (compiler option) ............... 247
guidelines, reading ................................ 21

H

Harbison, Samuel P. .................................. 25
hardware support in compiler ....................... 99
hash_map (STL header file) ........................ 394
hash_set (STL header file) ........................ 394
  __has_constructor, symbol used in library ........ 397
  __has_destructor, symbol used in library ........ 397
hdrstop (pragma directive) ........................ 475, 489
header files
  C .................................................. 391
  C++ .............................................. 392–393
  library ........................................... 389
  special function registers ...................... 214
  STL .............................................. 393
  DLib_Defaults.h .................................. 113, 118
  including stdbool.h for bool .................. 299
  including stddef.h for wchar_t ............... 300
header names, implementation-defined behavior .......... 473
--header_context (compiler option) ............... 247
heap
dynamic memory .................................. 55
storing data ....................................... 53
VLA allocated on .................................. 266
heap size
  and standard I/O ................................ 185
  changing default ................................ 90
HEAP (ELF section) ................................ 426
HEAP (section) .................................... 185
heap (zero-sized), implementation-defined behavior .... 479
hide (isymexport directive) ....................... 448
hints
  for good code generation ..................... 211
  implementation-defined behavior ............. 472
  using efficient data types ................... 197

I

-I (compiler option) ................................ 247
identifiers, implementation-defined behavior in C89 .......................... 484
IEEE format, floating-point values .............................................. 304
elfdump ................................................................. 442
options summary .......................................................... 443
elftool ................................................................. 440
options summary .......................................................... 441
if (linker directive) .................................................... 421
--hex (elftool option) ................................................ 457
ILINK options. See linker options
ILINK. See linker
--image_input (linker option) ........................................ 282
implementation-defined behavior
C89 ................................................................. 483
Standard C ........................................................... 467
important typedef (pragma directive) ...................................... 475, 489
include files
  including before source files ........................................... 260
  specifying .......................................................... 221
include (linker directive) ................................................ 421
include_alias (pragma directive) ...................................... 331
infinity ............................................................... 305
infinity (style for printing), implementation-defined
behavior ................................................................. 478
inheritance, in Embedded C++ ............................................. 171
initialization
  changing default ..................................................... 90
  C++ dynamic ......................................................... 78
  dynamic ............................................................ 113
  manual .............................................................. 91
  packing algorithm for ............................................... 91
  single-value ...................................................... 168
  suppressing ........................................................ 90
initialize (linker directive) ............................................. 408
initializers, static ..................................................... 167
-init_array (section) ................................................ 426
--inline (linker option) .............................................. 283
inline assembler ........................................................ 139
avoiding ............................................................... 211
See also assembler language interface

inline functions .......................................................... 161
  in compiler .......................................................... 296
  inline (pragma directive) ......................................... 332
  inlining functions, implementation-defined behavior ............ 472
  installation directory ............................................. 26
  instantiate (pragma directive) .................................... 475, 489
  instruction scheduling (compiler option) ......................... 210
  int (data type) signed and unsigned ................................ 299
  integer types ....................................................... 298
    casting .......................................................... 306
    implementation-defined behavior ................................ 470
    intptr_t .......................................................... 306
    ptrdiff_t ........................................................ 306
    size_t ............................................................ 306
    uintptr_t ........................................................ 307
  integers, implementation-defined behavior in C89 ................ 485
    integral promotion ................................................. 213
    Intel hex .......................................................... 183
    Intel IA64 ABI .................................................. 192
    internal error .................................................... 226
    __interrupt (extended keyword) .................................. 60
    interrupt functions .............................................. 59
      fast interrupts ............................................... 61
      in ARM Cortex .................................................. 64
      nested interrupts ............................................. 61
      operations ...................................................... 63
      software interrupts .......................................... 62
    interrupt handler. See interrupt service routine
    interrupt service routine ........................................ 59
    interrupt state, restoring ...................................... 367
    interrupt vector table .......................................... 64
    start address for ............................................... 59
    interrupts
      processor state ................................................. 54
      using with EC++ destructors .................................... 177
      __interwork (extended keyword) ................................ 318
    --interwork (compiler option) .................................. 248
    intptr_t (integer type) ......................................... 306
    __intrinsic (extended keyword) ................................ 318

See IAR C/C++ Development Guide
506  Compiling and Linking for ARM
intrinsic functions ........................................... 211
for Neon ..................................................... 351
overview .................................................... 137
summary ................................................... 345
intrinsics.h (header file) ................................. 345
inttypes.h (library header file) ......................... 392
.intvec (ELF section) .................................... 426
invocation syntax ........................................ 219
iojmanip ................................................... 443
options summary .......................................... 444
iomanip (library header file) ......................... 393
ios (library header file) ................................. 393
iosfwd (library header file) ............................ 393
__irq (extended keyword) .............................. 318
IRQ_STACK (section) .................................... 427
__JSB (intrinsinc function) ............................ 357
iso646.h (library header file) ......................... 392
istream (library header file) ......................... 393
options summary .......................................... 446
italic style, in this guide ............................... 26
iterator (STL header file) ............................... 394
I/O register. See SFR

J
Josuttis, Nicolai M. ........................................ 25

K
--keep (linker option) .................................... 283
keep (linker directive) .................................. 411
keep_definition (pragma directive) ..................... 475, 489
Kernighan & Ritchie function declarations ............ 212
disallowing ............................................... 262
Kernighan, Brian W. ...................................... 25
keywords ................................................... 313
extended, overview of ................................ 35

L
-l (compiler option) ....................................... 248
for creating skeleton code .............................. 148
labels ....................................................... 168
assembler, making public ................................ 261
__iar_program_start ..................................... 114
Labrosse, Jean J. ........................................ 25
Lajoie, Joséé ............................................... 25
language extensions
Embedded C++ ........................................... 171
enabling using pragma .................................. 332
enabling (-e) ............................................. 242
language overview ....................................... 33
language (pragma directive) ......................... 332
__LDC (intrinsinc function) ......................... 357
__LDC (intrinsinc function) ............................ 357
__LDC_noidx (intrinsinc function) ..................... 358
__LDC_noidx (intrinsinc function) ..................... 358
__LDC2 (intrinsinc function) ......................... 357
__LDC2L (intrinsinc function) ......................... 357
__LDC2L_noidx (intrinsinc function) .................. 358
__LDC2L_noidx (intrinsinc function) .................. 358
__LDC2_noidx (intrinsinc function) ................... 358
__LDC2_noidx (intrinsic function) .................... 358
__LDREX (intrinsinc function) ......................... 359
__LDREXB (intrinsic function) ......................... 359
__LDREXD (intrinsic function) ......................... 359
__LDREXH (intrinsic function) ......................... 359
--legacy (compiler option) ........................... 249
Lempel-Ziv-Welch algorithm, for packing initializers . 409
libraries
reason for using ......................................... 40
standard template library ............................ 393
using a prebuilt ...................................... 101
library configuration files
DLib ...................................................... 118
DLib_Defaults.h ......................................... 118, 119
modifying ............................................... 113
specifying .............................................. 242
library documentation .................................. 389

Part 2. Reference information 507
low_level_init.s99......................... 113
low-level processor operations............ 162, 345
accessing.................................. 137
__mcr (library function).................. 123
lzw, packing algorithm for initializers... 409
lz77, packing algorithm for initializers... 409

M

macros
  embedded in #pragma optimize................ 336
  ERANGE (in erro.h) ................. 476, 490
  inclusion of assert.................... 388
  NULL, implementation-defined behavior.... 477
    in C89 for DLIB...................... 490
  substituted in #pragma directives....... 162
  variadic.................................. 161
#pragma_positions_in_diagnostics (compiler option) .... 250
main (function)
  definition (C89) ....................... 483
  implementation-defined behavior........ 468
malloc (library function)
  See also heap ......................... 55
  implementation-defined behavior in C89.. 492
  -mangled_names_in_messages (linker option).... 285
Mann, Bernhard............................ 25
-map (linker option) ...................... 285
map file, producing ...................... 285
map (STL header file)..................... 394
math functions rounding mode,
  implementation-defined behavior........ 480
math functions (library functions)......... 127
math.h (library header file)............. 392
max recursion depth (stack usage control directive)..... 432
mb_len_max, implementation-defined behavior..... 480
__mcr (intrinsic function)................ 359
__mcr2 (intrinsic function)............... 359
memory
  allocating in C++........................ 55
dynamic.................................... 55
heap........................................ 55
non-initialized............................ 215
RAM, saving................................ 211
releasing in C++........................... 55
stack....................................... 54
  saving.................................... 211
used by global or static variables.......... 53
memory clobber............................. 140
memory layout, ARM........................ 53
memory management, type-safe.............. 171
memory map, output from linker............... 223
memory (pragma directive)................ 475, 489
memory (STL header file).................. 394
__memory_of, symbol used in library........ 397
-merge_duplicate_sections (linker option) .... 286
message (pragma directive)................ 334
messages
  disabling............................... 264, 293
  forcing.................................. 334
Meyers, Scott............................. 25
 pragma_positions_in_diagnostics (compiler option) .... 250
migration, from earlier IAR compilers........ 24
MISRA C, documentation................... 24
  --misra (compiler option).............. 231
  --misra (linker option)................. 270
  --misra1998 (compiler option).......... 270
  --misra1998 (linker option)............ 270
  --misra2004 (compiler option).......... 270
  --misra2004 (linker option)............ 270
mode changing, implementation-defined behavior..... 478
module consistency........................ 135
rtmodel.................................... 339
modules, introduction..................... 70
module_name (pragma directive)........... 475, 489
module-spec (in stack usage control file).... 433
Motorola S-records........................ 183
Part 2. Reference information

Index

pointer constant, relaxation to Standard C .................................. 166
numbers (in linker configuration file) ........................................ 420
numeric conversion functions,
implementation-defined behavior ......................................... 482
numeric (STL header file) ..................................................... 394
-O (compiler option) .......................................................... 258
-o (compiler option) .......................................................... 259
-o (iarchive option) ......................................................... 458
-o (ielfdump option) ......................................................... 458
-o (linker option) ............................................................ 290
object attributes ............................................................. 315
object filename, specifying (-o) ............................................. 259
object files, linker search path to (--search) ............................ 292
object_attribute (pragma directive) ....................................... 215
--only_stdout (compiler option) ......................................... 259
--only_stdout (linker option) ............................................. 290
__open (library function) ................................................... 123
operators
   See also @ (operator)
for cast
   in Extended EC++ ......................................................... 172
   missing from Embedded C++ ........................................... 172
for region expressions ...................................................... 404
for section control .......................................................... 165
precision for 32-bit float ................................................ 304
precision for 64-bit float ................................................ 305
sizeof, implementation-defined behavior ................................ 481
variants for cast ............................................................ 179
__Pragma (preprocessor) .................................................... 161
__ALIGNOF__, for alignment control ..................................... 164
?, language extensions for ................................................. 180
optimization
clustering, disabling ....................................................... 251
code motion, disabling .................................................... 251
common sub-expression elimination, disabling ....................... 252
configuration ............................................................... 49
disabling ................................................................. 208
function inlining, disabling (--no_inline) ............................ 253
hints ................................................................. 211
loop unrolling, disabling ................................................ 257
scheduling, disabling ..................................................... 255
specifying (-O) ............................................................ 258
--alignof, for alignment control ........................................ 208
type-based alias analysis, disabling (--tbaa) ......................... 256
using inline assembler code .............................................. 140
using pragma directive ................................................... 335
optimization levels ......................................................... 207
optimize (pragma directive) .............................................. 335
option parameters .......................................................... 227
options, compiler. See compiler options
options, iarchive. See iarchive options
options, ielfdump. See ielfdump options
options, ielftool. See ielftool options
options, iobjmanip. See iobjmanip options
options, isymexport. See isymexport options
options, linker. See linker options
--option_name (compiler option) ...................................... 278
Oram, Andy ................................................................. 25
ostream (library header file) ............................................. 393
output
   from preprocessor ...................................................... 260
   specifying for linker ................................................ 47
--output (compiler option) ............................................. 259
--output (iarchive option) ............................................. 458
--output (ielfdump option) ............................................ 458
--output (linker option) .............................................. 290
overhead, reducing ....................................................... 208–209
Pack (pragma directive) ................................................... 307, 337
packbits, packing algorithm for initializers ............................ 409
__packed (extended keyword) .......................................... 320
packed structure types .................................................... 307
Printf (library function) ......................... 105
choosing formatter ................................... 105
configuration symbols ............................... 121
implementation-defined behavior .................... 479
implementation-defined behavior in C89 .......... 492
__printf_args (pragma directive) .................. 337
printing characters, implementation-defined behavior ... 481
processor operations
accessing ............................................. 137
low-level ............................................. 162, 345
program entry label .................................... 114
program termination, implementation-defined behavior ... 468
programming hints .................................... 211
projects
basic settings for ..................................... 48
setting up for a library ............................... 112
prototypes, enforcing .................................. 262
ptrdiff_t (integer type) .............................. 306
PUBLIC (assembler directive) ....................... 261
publication date, of this guide ...................... 2
--public_equ (compiler option) ..................... 475, 490
public_equ (pragma directive) ..................... 126
putenv (library function), absent from DLIB ........ 126
putw, in stdio.h ..................................... 396
Q
__QADD (intrinsic function) ......................... 362
__QADDD8 (intrinsic function) ...................... 362
__QADDD16 (intrinsic function) .................... 363
__QASX (intrinsic function) ......................... 363
QCCARM (environment variable) ..................... 220
__QCCFlag (intrinsic function) ..................... 363
__QDADD (intrinsic function) ......................... 362
__QDOUBLE (intrinsic function) ..................... 363
__QDADD16 (intrinsic function) ...................... 363
__QDADD8 (intrinsic function) ....................... 362
__QFlag (intrinsic function) ......................... 364
__QSAX (intrinsic function) ......................... 363
__QSUB (intrinsic function) ......................... 362
__QSUB16 (intrinsic function) ....................... 363
__QSUB8 (intrinsic function) ....................... 363
qualifiers
const and volatile ..................................... 309
implementation-defined behavior .................. 473
implementation-defined behavior in C89 .......... 487
queue (STL header file) ............................. 394
R
-ra (compiler option) ................................ 238
-ra (archive option) .................................. 461
raise (library function), configuring support for .... 127
raise.c ............................................. 127
RAM
example of declaring region ......................... 74
execution ............................................. 58
initializers copied from ROM ....................... 46
running code from .................................... 93
saving memory ........................................ 211
__ramfunc (extended keyword) ....................... 321
--ram_reserve_ranges (isymexport option) ......... 459
range errors ......................................... 95
--raw (ielfdump option) ............................. 459
__RBIT (intrinsic function) ......................... 364
__read (library function) ............................ 123
customizing .......................................... 119
read formatter, selecting ......................... 107
reading guidelines .................................... 21
reading, recommended ................................ 25
realloc (library function) ........................... 137
implementation-defined behavior in C89 .......... 492
See also heap
recursive functions
avoiding ............................................. 211
storing data on stack ................................ 54
--redirect (linker option) ......................... 291
reentrancy (DLIB) .................................... 390

Part 2. Reference information 513
reference information, typographic convention ........................................ 26
region expression (in linker configuration file) ........................................ 403
region literal (in linker configuration file). ............................................ 402
register keyword, implementation-defined behavior ................................. 472
register parameters .................................................................................. 152–153
registered trademarks ............................................................................... 2
registers
  assigning to parameters ................................................................. 153
  callee-save, stored on stack ............................................................ 54
  for function returns ......................................................................... 154
  implementation-defined behavior in C89 .......................................... 487
in assembler-level routines ................................................................. 150
  preserved ......................................................................................... 152
  scratch ............................................................................................. 152
reinterpret_cast (cast operator) ............................................................ 172
  .rel (ELF section) ........................................................................... 424
  .rela (ELF section) ......................................................................... 424
  --relaxed_fp (compiler option) ..................................................... 261
relay, see vencers .................................................................................... 95
relocation errors, resolving ................................................................. 95
remark (diagnostic message) ................................................................. 225
  classifying for compiler ................................................................. 240
  classifying for linker ................................................................. 276
  enabling in compiler ................................................................. 262
  enabling in linker ................................................................. 292
  --remarks (compiler option) .......................................................... 262
  --remarks (linker option) .............................................................. 292
remove (library function) ................................................................. 123
  implementation-defined behavior ................................................. 478
  implementation-defined behavior in C89 (DLIB) ......................... 491
  --remove_section (iobjmanip option) ........................................... 460
remquo, magnitude of ................................................................. 477
rename (isymexport directive) ............................................................. 448
  library function ............................................................................ 123
  implementation-defined behavior ................................................. 478
  implementation-defined behavior in C89 (DLIB) ......................... 492
  --rename_section (iobjmanip option) ............................................ 460
  --rename_symbol (iobjmanip option) ............................................. 461
  --replace (iarchive option) .......................................................... 461
required (pragma directive) ................................................................. 338
  --require_prototypes (compiler option) ....................................... 262
  --reserve_ranges (isymexport option) ........................................... 462
reset vector table ............................................................................... 426
  __reset_Q_flag (intrinsic function) ................................................. 364
  __reset_QC_flag (intrinsic function) .............................................. 364
return values, from functions ........................................................... 154
  __REV (intrinsic function) ............................................................. 365
  __REVS (intrinsic function) .......................................................... 365
  __REV16 (intrinsic function) .......................................................... 365
Ritchie, Dennis M. ............................................................................ 25
.rodata (ELF section) ........................................................................... 427
ROM to RAM, copying ......................................................................... 93
  __root (extended keyword) .......................................................... 321
  __ROPI__ (predefined symbol) ...................................................... 386
  --ropi (compiler option) .............................................................. 262
routines, time-critical .......................................................................... 137, 162, 345
rtmodel (assembler directive) ............................................................ 136
rtmodel (pragma directive) ................................................................. 338
RTTI data (dynamic), including in the image ..................................... 286
rtti support, missing from STL .......................................................... 172
runtime environment
  DLIB .............................................................................................. 99
  setting options for ........................................................................ 51
  setting up (DLIB) ........................................................................ 100
runtime libraries (DLIB)
  introduction ................................................................................... 389
  customizing system startup code .................................................. 117
  customizing without rebuilding ...................................................... 104
  filename syntax ........................................................................... 102
  overriding modules in ..................................................................... 111
  using prebuilt ............................................................................... 101
runtime library
  setting up from command line ......................................................... 50
  setting up from IDE ....................................................................... 50
runtime model attributes ........................................................................ 135
runtime model definitions ...................................................................... 339
runtime type information, missing from Embedded C++ .......................... 172
  __RWPI__ (predefined symbol) ...................................................... 387

IAR C/C++ Development Guide
514 Compiling and Linking for ARM
--rwpi (compiler option) ........................................... 263

S

-S (iarchive option) ............................................. 463
-s (ielfdump option) .......................................... 462
__SADD8 (intrinsic function) .................................. 365
__SADD16 (intrinsic function) ................................ 365
__SASX (intrinsic function) .................................... 365
__sbrel (extended keyword) .................................... 316
scanf (library function)
  choosing formatter (DLIB) ...................................... 106
  configuration symbols ........................................ 121
  implementation-defined behavior .................................. 479
  implementation-defined behavior in C89 (DLIB) ................... 492
__scanf_args (pragma directive) .................................. 339
scheduling (compiler transformation)
  disabling ......................................................... 210
scratch registers .................................................. 152
--search (linker option) ........................................ 292
search path to library files (--search) .......................... 292
search path to object files (--search) .......................... 292
--section (ielfdump option) .................................... 462
--section (compiler option) ..................................... 263
sections .......................................................... 423
  summary ......................................................... 423
  allocation of ................................................ 73
  declaring (#pragma section) .................................... 340
  introduction .................................................. 70
  specifying (--section) ........................................... 263
__section_begin (extended operator) ............................. 165
__section_end (extended operator) .............................. 165
__section_size (extended operator) .............................. 165
segment (pragma directive) ...................................... 340
segments, declaring (#pragma segment) .......................... 340
__SEL (intrinsic function) ...................................... 365
--self_reloc (ielftool option) .................................. 463
--semihosting (linker option) .................................. 292
semihosting, overview ........................................ 108

--separate_cluster_for_initialized_variables
(compiler option) ................................................ 264
set (STL header file) ............................................ 264
setjmp.h (library header file) .................................. 392
setlocale (library function) .................................... 125
settings, basic for project configuration ....................... 48
__set_BASEPRI (intrinsic function) .............................. 366
__set_CONTROL (intrinsic function) ............................. 366
__set_CPSR (intrinsic function) ................................ 366
__set_FAULTMASK (intrinsic function) .......................... 366
__set_FPSCR (intrinsic function) ................................ 366
__set_interrupt_state (intrinsic function) ...................... 367
__set_LR (intrinsic function) .................................... 367
__set_MSP (intrinsic function) ................................... 367
__set_PRIMASK (intrinsic function) .............................. 367
__set_PSP (intrinsic function) .................................. 367
__set_SB (intrinsic function) .................................... 368
__set_SP (intrinsic function) .................................... 368
__SEV (intrinsic function) ....................................... 368
severity level, of diagnostic messages .......................... 225
  specifying ..................................................... 226
SFR
  accessing special function registers ......................... 214
  declaring extern special function registers .................. 203
__SHAD8 (intrinsic function) .................................... 368
__SHADD16 (intrinsic function) ................................ 368
shared object .................................................. 222, 285
__SHASX (intrinsic function) .................................... 368
short (data type) ................................................ 299
show (symexport directive) ...................................... 448
__SHASX (intrinsic function) .................................... 368
__shstrtab (ELF section) ......................................... 424
__SHSUB16 (intrinsic function) ................................ 368
__SHSUB8 (intrinsic function) ................................... 368
signal (library function)
  configuring support ........................................... 127
  implementation-defined behavior ............................... 477
  implementation-defined behavior in C89 ......................... 490
signals, implementation-defined behavior ....................... 468
  at system startup ............................................ 468
signal.c .............................................. 127
signal.h (library header file) .................. 392
signed char (data type) ......................... 298–299
specifying ........................................ 235
signed int (data type) .......................... 299
signed long long (data type) .................... 299
signed long (data type) .......................... 299
signed short (data type) ......................... 299
--silent (compiler option) ..................... 264
--silent (archiver option) ..................... 463
--silent (elftool option) ...................... 463
--silent (linker option) ....................... 293
silent operation
specifying in compiler .......................... 264
specifying in linker ............................. 293
--simple (elftool option) ..................... 464
sin (library function) ........................... 128
sinf (library function) ......................... 128
size (in stack usage control file) ............ 435
size_t (integer type) ............................ 306
skeleton code, creating for assembler language interface: 147
skeleton.s (assembler source output) ........ 148
--skip_dynamic_initialization (linker option) ... 293
slist (STL header file) ......................... 394
smallest, packing algorithm for initializers ... 409
__SMLABBB (intrinsic function) ............... 369
__SMLABBT (intrinsic function) ............... 369
__SMLAD (intrinsic function) .................. 369
__SMLADX (intrinsic function) ............... 369
__SMLALBB (intrinsic function) ............... 370
__SMLALBT (intrinsic function) ............... 370
__SMLALD (intrinsic function) .................. 370
__SMLALDX (intrinsic function) ............... 370
__SMLALTBB (intrinsic function) ............... 370
__SMLATBT (intrinsic function) ............... 370
__SMLATD (intrinsic function) .................. 370
__SMLATDX (intrinsic function) ............... 370
__SMLAWBB (intrinsic function) ............... 369
__SMLAWBT (intrinsic function) ............... 369
__SMLAWD (intrinsic function) .................. 369
__SMLAWDX (intrinsic function) ............... 369
__SMLSLBB (intrinsic function) ............... 370
__SMLSLBT (intrinsic function) ............... 370
__SMLSLD (intrinsic function) .................. 370
__SMLSLDX (intrinsic function) ............... 370
__SMLSLTBB (intrinsic function) ............... 370
__SMLSTBT (intrinsic function) ............... 370
__SMLSTD (intrinsic function) .................. 370
__SMLSTDX (intrinsic function) ............... 370
__SMLSTTBB (intrinsic function) ............... 370
__SMLSTTBT (intrinsic function) ............... 370
__SMLSTTD (intrinsic function) .................. 370
__SMLSTTDX (intrinsic function) ............... 370
__SMLTBB (intrinsic function) .................. 370
__SMLTBT (intrinsic function) .................. 370
__SMLTD (intrinsic function) .................. 370
__SMLTDX (intrinsic function) .................. 370
__SMLT (intrinsic function) .................. 370
__SMLS (intrinsic function) .................. 370
__SMLSD (intrinsic function) .................. 370
__SMLSDX (intrinsic function) ............... 370
__SML (intrinsic function) .................. 370
__SMULBB (intrinsic function) ............... 371
__SMULBT (intrinsic function) ............... 371
__SMULTB (intrinsic function) ............... 371
__SMULT (intrinsic function) .................. 371
__SMUL (intrinsic function) .................. 371
__SMULWB (intrinsic function) ............... 372
__SMULWT (intrinsic function) ............... 372
__SMUSD (intrinsic function) .................. 371
__SMUSDX (intrinsic function) ............... 371
__SMUAD (intrinsic function) ............... 371
__SMUL (intrinsic function) .................. 372
__SMULBB (intrinsic function) ............... 372
__SMULBT (intrinsic function) ............... 372
__SMULTB (intrinsic function) ............... 372
__SMULT (intrinsic function) .................. 372
__SMULWB (intrinsic function) ............... 372
__SMULWT (intrinsic function) ............... 372
special function registers (SFR) .............. 214
special function types .......................... 59
overview ................................ .......... 35
sprintf (library function) ................... 105
choosing formatter .............................. 105
--src (elftool option) ....................... 464
--srec (elftool option) ...................... 464
--srec-len (elftool option) .................. 464
--srec-only (elftool option) ............... 465
__SSAT (intrinsic function) .................. 372
__SSAT16 (intrinsic function) ............... 373
__SSAX (intrinsic function) ............... 365
sscanf (library function), choosing formatter (DLIB) .... 106
sstream (library header file) ............... 393
IAR C/C++ Development Guide

Compiling and Linking for ARM

518

Compiling and Linking for ARM

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199

199
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tanl (library routine)</td>
<td>128-129</td>
</tr>
<tr>
<td>__task (extended keyword)</td>
<td>323</td>
</tr>
<tr>
<td>technical support, IAR Systems</td>
<td>226</td>
</tr>
<tr>
<td>template support</td>
<td>172, 178</td>
</tr>
<tr>
<td>in Extended EC++</td>
<td>171</td>
</tr>
<tr>
<td>missing from Embedded C++</td>
<td>171</td>
</tr>
<tr>
<td>Terminal I/O window</td>
<td>109</td>
</tr>
<tr>
<td>making available (DLIB)</td>
<td>112</td>
</tr>
<tr>
<td>terminal I/O, debugger runtime interface for</td>
<td>108</td>
</tr>
<tr>
<td>terminal output, speeding up</td>
<td>109</td>
</tr>
<tr>
<td>termination of system. See system termination</td>
<td>26</td>
</tr>
<tr>
<td>termination status, implementation-defined behavior</td>
<td>479</td>
</tr>
<tr>
<td>terminology</td>
<td>132</td>
</tr>
<tr>
<td>.text (ELF section)</td>
<td>324</td>
</tr>
<tr>
<td>tgmath.h (library header file)</td>
<td>428</td>
</tr>
<tr>
<td>32-bits (floating-point format)</td>
<td>304</td>
</tr>
<tr>
<td>this (pointer)</td>
<td>149</td>
</tr>
<tr>
<td>threaded environment</td>
<td>130</td>
</tr>
<tr>
<td>thread-local storage</td>
<td>132</td>
</tr>
<tr>
<td>__thumb (extended keyword)</td>
<td>324</td>
</tr>
<tr>
<td>--thumb (compiler option)</td>
<td>265</td>
</tr>
<tr>
<td><strong>TIME</strong> (predefined symbol)</td>
<td>387</td>
</tr>
<tr>
<td>time zone (library function)</td>
<td>493</td>
</tr>
<tr>
<td>implementation-defined behavior in C89</td>
<td>493</td>
</tr>
<tr>
<td>time zone (library function), implementation-defined behavior</td>
<td>480</td>
</tr>
<tr>
<td>time-critical routines</td>
<td>137, 162, 345</td>
</tr>
<tr>
<td>time.c (library header file)</td>
<td>127</td>
</tr>
<tr>
<td>time.h (library header file)</td>
<td>392</td>
</tr>
<tr>
<td>additional C functionality</td>
<td>396</td>
</tr>
<tr>
<td>time32 (library function), configuring support for</td>
<td>127</td>
</tr>
<tr>
<td>time64 (library function), configuring support for</td>
<td>127</td>
</tr>
<tr>
<td>tips, programming</td>
<td>211</td>
</tr>
<tr>
<td>TLS handling</td>
<td>132</td>
</tr>
<tr>
<td>--toc (iarchive option)</td>
<td>466</td>
</tr>
<tr>
<td>tools icon, in this guide</td>
<td>27</td>
</tr>
<tr>
<td>trademarks</td>
<td>2</td>
</tr>
<tr>
<td>transformations, compiler</td>
<td>205</td>
</tr>
<tr>
<td>translation, implementation-defined behavior</td>
<td>467</td>
</tr>
<tr>
<td>translation, implementation-defined behavior in C89</td>
<td>483</td>
</tr>
<tr>
<td>type attributes</td>
<td>313</td>
</tr>
<tr>
<td>specifying</td>
<td>342</td>
</tr>
<tr>
<td>type qualifiers</td>
<td>309</td>
</tr>
<tr>
<td>const and volatile</td>
<td>309</td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>473</td>
</tr>
<tr>
<td>implementation-defined behavior in C89</td>
<td>487</td>
</tr>
<tr>
<td>typedefs</td>
<td>256</td>
</tr>
<tr>
<td>excluding from diagnostics</td>
<td>256</td>
</tr>
<tr>
<td>repeated</td>
<td>167</td>
</tr>
<tr>
<td>typeinfo (library header file)</td>
<td>393</td>
</tr>
<tr>
<td>type_attribute (pragma directive)</td>
<td>342</td>
</tr>
<tr>
<td>type-based alias analysis (compiler transformation)</td>
<td>209</td>
</tr>
<tr>
<td>disabling</td>
<td>256</td>
</tr>
<tr>
<td>type-safe memory management</td>
<td>171</td>
</tr>
<tr>
<td>typographic conventions</td>
<td>26</td>
</tr>
<tr>
<td>__UADD8 (intrinsic function)</td>
<td>376</td>
</tr>
<tr>
<td>__UADD16 (intrinsic function)</td>
<td>376</td>
</tr>
<tr>
<td>__UASX (intrinsic function)</td>
<td>376</td>
</tr>
<tr>
<td>uchar.h (library header file)</td>
<td>392</td>
</tr>
<tr>
<td>__UHADD8 (intrinsic function)</td>
<td>377</td>
</tr>
<tr>
<td>__UHADD16 (intrinsic function)</td>
<td>377</td>
</tr>
<tr>
<td>__UHASX (intrinsic function)</td>
<td>377</td>
</tr>
<tr>
<td>__UHASX (intrinsic function)</td>
<td>377</td>
</tr>
<tr>
<td>__UHSUB16 (intrinsic function)</td>
<td>377</td>
</tr>
<tr>
<td>__UHSUB8 (intrinsic function)</td>
<td>377</td>
</tr>
<tr>
<td>uintptr_t (integer type)</td>
<td>307</td>
</tr>
<tr>
<td>__UMAAL (intrinsic function)</td>
<td>377</td>
</tr>
<tr>
<td>underflow errors, implementation-defined behavior</td>
<td>476</td>
</tr>
<tr>
<td>underflow range errors</td>
<td>476</td>
</tr>
<tr>
<td>implementation-defined behavior in C89</td>
<td>490</td>
</tr>
<tr>
<td>__ungetchar, in stdio.h</td>
<td>396</td>
</tr>
<tr>
<td>unions</td>
<td>164, 199</td>
</tr>
<tr>
<td>anonymous</td>
<td>164, 199</td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>473</td>
</tr>
<tr>
<td>implementation-defined behavior in C89</td>
<td>487</td>
</tr>
</tbody>
</table>
universal character names, implementation-defined behavior .............................................. 474
unsigned char (data type) .......................................................... 298–299
  changing to signed char ........................................................... 235
unsigned int (data type) .......................................................... 299
unsigned long long (data type) .................................................... 299
unsigned long (data type) .......................................................... 299
unsigned short (data type) .......................................................... 299
  __UQADD8 (intrinsic function) .................................................... 378
  __UQADD16 (intrinsic function) ................................................... 378
  __UQASX (intrinsic function) ..................................................... 378
  __UQSAX (intrinsic function) ..................................................... 378–379
  __UQSUB16 (intrinsic function) ................................................... 378–379
  __UQSUB8 (intrinsic function) ..................................................... 378–379
  __USAD8 (intrinsic function) ..................................................... 378
  __USAD16 (intrinsic function) .................................................... 378
  __USAT (intrinsic function) ..................................................... 378
  __USAT16 (intrinsic function) .................................................... 379
  __USAX (intrinsic function) ..................................................... 378
  --use_c++ inline (compiler option) ............................................ 266
  --use_unix_directory_separators (compiler option) ...................... 266
  --USUB8 (intrinsic function) .................................................... 376
  --use_vfp (compiler option) .................................................... 246
  --vfe (compiler option) ......................................................... 294
  --vla (compiler option) ......................................................... 266
void, pointers to .............................................................. 166
volatile
  and const, declaring objects .................................................. 310
  declaring objects .................................................................. 309
  protecting simultaneously accesses variables, rules for access .......... 213

W

  #warning message (preprocessor extension) .................................. 388
  warnings .............................................................................. 225
    classifying in compiler .......................................................... 240
    classifying in linker ............................................................. 277
    disabling in compiler .......................................................... 258
    disabling in linker .............................................................. 289
    exit code in compiler .......................................................... 267
    exit code in linker .............................................................. 295
  warnings icon, in this guide ...................................................... 27
    warnings (pragma directive) ................................................... 475, 490
    --warnings_affect_exit_code (compiler option) ......................... 223, 267
    --warnings_affect_exit_code (linker option) ............................ 295
--warnings_are_errors (compiler option) .......................... 267
--warnings_are_errors (linker option) .......................... 295
wchar_t (data type), adding support for in C ................. 300
wchar.h (library header file) ................................... 392
__weak (extended keyword) ...................................... 324
weak (pragma directive) .......................................... 342
web sites, recommended ........................................... 25
__WFE (intrinsic function) ....................................... 380
__WFI (intrinsic function) ....................................... 380
white-space characters, implementation-defined behavior 467
__write (library function) ........................................ 123
customizing ....................................................... 119
__write_array, in stdio.h ....................................... 396
__write_buffered (DLIB library function) ....................... 109

X
-x (archive option) ................................................. 456
xreportassert.c ................................................... 129

Y
__YIELD (intrinsic function) ..................................... 380

Z
zeros, packing algorithm for initializers ....................... 409

Symbols
--bin (ielftool option) .......................... 451
--call_graph (compiler option) ................. 273
--char_is_unsigned (compiler option) ........... 235
--char_is_signed (compiler option) ............. 235
--checksum (ielftool option) .................... 451
--code (ielfdump option) ........................ 454
--config (linker option) ........................ 273
--config_def (linker option) .................... 274
--cpp_init_routine (linker option) .............. 274
--cpu (compiler option) .......................... 235
--cpu_mode (compiler option) ..................... 237
--create (iarchive option) ....................... 454
--c++ (compiler option) ........................... 237
--c89 (compiler option) .......................... 235
--debug (compiler option) ......................... 238
--delete (iararchive option) ...................... 455
--dependencies (compiler option) ............... 238
--dependencies (linker option) .................. 275
--diagnostics_tables (compiler option) ........ 241
--diagnostics_tables (linker option) .......... 278
--diag_error (compiler option) .................. 239
--diag_error (linker option) ..................... 276
--diag_remark (compiler option) ................ 240
--diag_remark (linker option) .................... 276
--diag_suppress (compiler option) .............. 240
--diag_suppress (linker option) ................ 277
--diag_warning (compiler option) ............... 240
--diag_warning (linker option) ................... 277
--discard_unused_publics (compiler option) .... 241
--dibh_config (compiler option) ................. 242
--ec++ (compiler option) ......................... 243
--edit (isymexport option) ....................... 455
--ec++ (compiler option) ........................... 243
--enable_hardware_workaround (compiler option) 244
--enable_hardware_workaround (linker option) ... 278
--enable_multithreads (compiler option) ....... 244
--endian (compiler option) ....................... 244
--entry (linker option) ........................... 279
--enum_is_int (compiler option) ................. 245
--error_limit (compiler option) ................ 245
--error_limit (linker option) .................... 279
--exception_tables (linker option) .............. 280
--export_builtin_config (linker option) ......... 280
--extract (iarchive option) ...................... 456
--extra_init (linker option) ..................... 281
--fill (ielftool option) .......................... 457
--force_exceptions (linker option) .............. 281
--force_output (linker option) ................... 282
--fpu (compiler option) ............................ 246
--guard_calls (compiler option) .................. 247
--header_context (compiler option) .............. 247
--hex (ielftool option) ........................... 457
--image_input (linker option) .................... 282
--inline (linker option) .......................... 283
--interwork (compiler option) .................... 248
--keep (linker option) ............................. 283
--legacy (compiler option) ....................... 249
--lock_regs (compiler option) .................... 249
--log (linker option) .............................. 284
--log_file (linker option) ....................... 284
--macro_positions_in_diagnostics (compiler option) 250
--mangled_names_in_messages (linker option) .... 285
--map (linker option) .............................. 285
--merge_duplicate_sections (linker option) ..... 286
--mfc (compiler option) ............................ 250
--misrac (compiler option) ....................... 231
--misrac (linker option) .......................... 270
--misrac_verbose (compiler option) .............. 270
--misrac (linker option) .......................... 231
--misrac1998 (compiler option) .................. 231
--misrac1998 (linker option) ..................... 231
--misrac1998 (linker option) ..................... 270
--misrac1998 (linker option) ..................... 270
--no_cluster (compiler option) ................... 251
--no_const_align (compiler option) ............... 251
--no_code_motion (compiler option) ............... 251
--no_const_align (compiler option) ............... 251
--no_cse (compiler option) ....................... 252
Numerics

32-bits (floating-point format) .......................... 304
64-bits (floating-point format) ......................... 305