IAR C/C++ Development Guide
Compiling and linking

for Advanced RISC Machines Ltd’s
ARM Microprocessor Family
COPYRIGHT NOTICE
Copyright © 1999–2010 IAR Systems AB.

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

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

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

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

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

EDITION NOTICE
Fifth edition: April 2010

Part number: DARM-5

This guide applies to version 5.5x of IAR Embedded Workbench® for ARM.


Internal reference: R12, M5.7.0, ISUD.
Brief contents

Tables .................................................................................................................. xxvii
Preface ................................................................................................................. xxix

Part 1. Using the build tools ................................................................. 1
Introduction to the IAR build tools ......................................................... 3
Developing embedded applications ....................................................... 9
Data storage ...................................................................................................... 25
Functions ............................................................................................................... 29
Linking using ILINK ........................................................................................ 39
Linking your application .............................................................................. 47
The DLIB runtime environment ............................................................... 59
Assembler language interface ................................................................. 89
Using C++ .......................................................................................................... 105
Application-related considerations ....................................................... 113
Efficient coding for embedded applications ........................................ 125

Part 2. Reference information .............................................................. 143
External interface details ........................................................................... 145
Compiler options ........................................................................................... 153
Linker options .................................................................................................. 189
Data representation .......................................................................................... 209
Compiler extensions ........................................................................................ 221
Extended keywords ....................................................................................... 233
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pragma directives</td>
<td>245</td>
</tr>
<tr>
<td>Intrinsic functions</td>
<td>259</td>
</tr>
<tr>
<td>The preprocessor</td>
<td>283</td>
</tr>
<tr>
<td>Library functions</td>
<td>291</td>
</tr>
<tr>
<td>The linker configuration file</td>
<td>299</td>
</tr>
<tr>
<td>Section reference</td>
<td>321</td>
</tr>
<tr>
<td>IAR utilities</td>
<td>325</td>
</tr>
<tr>
<td>Implementation-defined behavior</td>
<td>353</td>
</tr>
<tr>
<td>Index</td>
<td>365</td>
</tr>
</tbody>
</table>
Contents

Tables .................................................................................................................. xxvii
Preface ................................................................................................................. xxix

  Who should read this guide ............................................................ xxix
  How to use this guide ....................................................................... xxix
  What this guide contains ................................................................. xxx
  Other documentation ................................................................. xxxi
    Further reading .............................................................................. xxxii
  Document conventions ........................................................................xxxiii
    Typographic conventions .............................................................. xxxiii
    Naming conventions ....................................................................... xxxiv

Part 1. Using the build tools ........................................................................... 1

  Introduction to the IAR build tools ..................................................... 3

    The IAR build tools—an overview ................................................. 3
      IAR C/C++ Compiler ................................................................. 3
      IAR Assembler ........................................................................... 4
      The IAR ILINK Linker ............................................................... 4
      Specific ELF tools ........................................................................ 4
      External tools ............................................................................ 4

    IAR language overview ................................................................... 5

    Device support ............................................................................... 5
      Supported ARM devices ............................................................ 5
      Preconfigured support files ......................................................... 6
      Examples for getting started ......................................................... 6

    Special support for embedded systems .......................................... 6
      Extended keywords .................................................................... 6
     Pragma directives .......................................................................... 7
      Predefined symbols ................................................................... 7
      Special function types .............................................................. 7
      Accessing low-level features ....................................................... 7
Developing embedded applications ........................................................... 9

Developing embedded software using IAR build tools ......................... 9
- Mapping of internal and external memory ............................................. 9
- Communication with peripheral units ................................................. 10
- Event handling .................................................................................. 10
- System startup .................................................................................. 10
- Real-time operating systems .............................................................. 10
- Interoperability with other build tools .............................................. 11

The build process—an overview ........................................................... 11
- The translation process ...................................................................... 11
- The linking process .......................................................................... 12
- After linking ..................................................................................... 14

Application execution—an overview ..................................................... 14
- The initialization phase ...................................................................... 15
- The execution phase ......................................................................... 19
- The termination phase ...................................................................... 19

Basic project configuration ..................................................................... 19
- Processor configuration .................................................................... 20
- Optimization for speed and size ....................................................... 21
- Runtime environment ...................................................................... 22

Data storage ............................................................................................. 25

Introduction ............................................................................................ 25
- Different ways to store data .............................................................. 25

Auto variables—on the stack ................................................................. 26
- The stack .......................................................................................... 26

Dynamic memory on the heap ............................................................... 27

Functions ................................................................................................ 29

Function-related extensions ................................................................... 29

ARM and Thumb code .......................................................................... 29

Execution in RAM .................................................................................. 30
Primitives for interrupts, concurrency, and OS-related programming ............................................................ 30
  Interrupt functions ............................................................................................................................... 31
  Installing exception functions ............................................................................................................ 31
  Interrupts and fast interrupts ............................................................................................................. 32
  Nested interrupts ............................................................................................................................... 33
  Software interrupts ............................................................................................................................ 34
  Interrupt operations ............................................................................................................................ 35
  Interrupts for ARM Cortex-M ............................................................................................................. 36
  C++ and special function types ........................................................................................................ 36

Linking using ILINK ........................................................................................................................... 39
  Linking—an overview ......................................................................................................................... 39
  Modules and sections ......................................................................................................................... 40
  The linking process ............................................................................................................................ 41
  Placing code and data—the linker configuration file ...................................................................... 42
    A simple example of a configuration file ......................................................................................... 43
  Initialization at system startup ........................................................................................................ 45
    The initialization process ................................................................................................................ 46

Linking your application ..................................................................................................................... 47
  Linking considerations ....................................................................................................................... 47
    Choosing a linker configuration file ............................................................................................... 47
    Defining your own memory areas ................................................................................................. 48
    Placing sections .............................................................................................................................. 49
    Reserving space in RAM ................................................................................................................ 50
    Keeping modules ............................................................................................................................. 51
    Keeping symbols and sections ........................................................................................................ 51
    Application startup .......................................................................................................................... 52
    Setting up the stack .......................................................................................................................... 52
    Setting up the heap .......................................................................................................................... 52
    Setting up the atexit limit ............................................................................................................... 52
    Changing the default initialization ............................................................................................... 52
    Interaction between ILINK and the application .......................................................................... 55
    Standard library handling .............................................................................................................. 56
Producing other output formats than ELF/DWARF .................................. 56
Veneers ........................................................................................................ 56

Hints for troubleshooting ........................................................................... 56
Relocation errors ........................................................................................ 57

The DLIB runtime environment ................................................................ 59

Introduction to the runtime environment .................................................. 59
Runtime environment functionality ............................................................. 59
Library selection .......................................................................................... 60
Situations that require library building ......................................................... 61
Library configurations .................................................................................. 61
Low-level interface for debug support .......................................................... 62

Using a prebuilt library ................................................................................. 62
Groups of library files .................................................................................... 63
Customizing a prebuilt library without rebuilding .......................................... 64

Choosing formatters for printf and scanf .................................................. 65
Choosing printf formatter ........................................................................... 65
Choosing scanf formatter .......................................................................... 66

Overriding library modules ......................................................................... 67

Building and using a customized library ...................................................... 68
Setting up a library project .......................................................................... 69
Modifying the library functionality .............................................................. 69
Using a customized library .......................................................................... 70

System startup and termination ................................................................. 70
System startup ............................................................................................ 70
System termination ....................................................................................... 73

Customizing system initialization ................................................................ 74
__low_level_init .......................................................................................... 74
Modifying the file cstartup.s ....................................................................... 75

Standard streams for input and output ....................................................... 75
Implementing low-level character input and output ...................................... 75

Configuration symbols for printf and scanf .............................................. 77
Customizing formatting capabilities ........................................................... 78

File input and output .................................................................................... 78
Locale ........................................................................................................... 79
  Locale support in prebuilt libraries ........................................................ 79
  Customizing the locale support ......................................................... 80
  Changing locales at runtime ............................................................. 81
Environment interaction ....................................................................... 81
  The getenv function ........................................................................... 82
  The system function ........................................................................... 82
Signal and raise .................................................................................... 82
Time ............................................................................................................. 83
Strtod ........................................................................................................... 83
Assert ........................................................................................................... 83
Atexit ........................................................................................................... 84
C-SPY runtime interface .................................................................... 84
  Low-level debugger runtime interface ............................................. 84
  The debugger terminal I/O window ................................................ 85
Checking module consistency ............................................................ 85
  Runtime model attributes ............................................................... 86
  Using runtime model attributes ..................................................... 86
Assembler language interface ............................................................. 89
  Mixing C and assembler ...................................................................... 89
    Intrinsic functions ........................................................................... 89
    Mixing C and assembler modules ................................................ 90
    Inline assembler ............................................................................ 91
  Calling assembler routines from C .................................................. 92
    Creating skeleton code ................................................................... 92
    Compiling the code ....................................................................... 93
  Calling assembler routines from C++ ................................................. 94
  Calling convention ............................................................................ 95
    Function declarations ..................................................................... 96
    Using C linkage in C++ source code ............................................. 96
    Preserved versus scratch registers .............................................. 96
    Function entrance ......................................................................... 97
    Function exit ................................................................................ 99
<table>
<thead>
<tr>
<th>AEABI compliance</th>
<th>Linking AEABI-compliant modules using the IAR ILINK Linker</th>
<th>Linking AEABI-compliant modules using a third-party linker</th>
<th>Enabling AEABI compliance in the compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>121</td>
<td>122</td>
<td>122</td>
</tr>
</tbody>
</table>

**Efficient coding for embedded applications** ........................................ 125

<table>
<thead>
<tr>
<th>Selecting data types</th>
<th>Using efficient data types</th>
<th>Floating-point types</th>
<th>Alignment of elements in a structure</th>
<th>Anonymous structs and unions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>125</td>
<td>126</td>
<td>127</td>
<td>127</td>
</tr>
</tbody>
</table>

**Controlling data and function placement in memory** .............. 129

<table>
<thead>
<tr>
<th>Data placement at an absolute location</th>
<th>Data and function placement in sections</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>130</td>
</tr>
</tbody>
</table>

**Controlling compiler optimizations** ............................................. 132

<table>
<thead>
<tr>
<th>Scope for performed optimizations</th>
<th>Optimization levels</th>
<th>Speed versus size</th>
<th>Fine-tuning enabled transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>132</td>
<td>133</td>
<td>134</td>
</tr>
</tbody>
</table>

**Facilitating good code generation** .............................................. 137

<table>
<thead>
<tr>
<th>Writing optimization-friendly source code</th>
<th>Saving stack space and RAM memory</th>
<th>Function prototypes</th>
<th>Integer types and bit negation</th>
<th>Protecting simultaneously accessed variables</th>
<th>Accessing special function registers</th>
<th>Passing values between C and assembler objects</th>
<th>Non-initialized variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>137</td>
<td>138</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
<td>141</td>
</tr>
</tbody>
</table>

**Part 2. Reference information** ....................................................... 143

**External interface details** ............................................................ 145

**Invocation syntax** ............................................................... 145

<table>
<thead>
<tr>
<th>Compiler invocation syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
</tr>
</tbody>
</table>
--discard_unused_publics ................................................................. 166
--dlib_config .................................................................................... 166
-e ...................................................................................................... 167
-ec++ ............................................................................................... 167
-ec++ ............................................................................................. 167
--enable_hardware_workaround ........................................... 168
--enable_multibytes ................................................................. 168
--endian ......................................................................................... 168
--enum_is_int ................................................................................... 169
--error_limit ...................................................................................... 169
-f ....................................................................................................... 169
--fpu .................................................................................................. 170
--header_context ............................................................................... 171
-I ....................................................................................................... 171
--interwork ....................................................................................... 171
-l ....................................................................................................... 172
--legacy ............................................................................................. 173
--mfc ................................................................................................. 173
--migration_preprocessor_extensions .............................................. 173
--no_clustering .................................................................................. 174
--no_code_motion ............................................................................ 174
--no_const_align ............................................................................. 175
--no_cse ........................................................................................... 175
--no_fragments ............................................................................... 175
--no_guard_calls ............................................................................. 176
--no_inline ....................................................................................... 176
--no_path_in_file_macros .................................................................. 176
--no_rwlock_dynamic_init .................................................................. 177
--no_scheduling ............................................................................... 177
--no_tbaa ......................................................................................... 178
--no_typedefs_in_diagnostics .......................................................... 178
--no_unaligned_access ..................................................................... 179
--no_unroll ....................................................................................... 179
--no_warnings .................................................................................. 180
--no_wrap_diagnostics ................................................................. 180
-O ............................................................................................... 180
-o, --output .............................................................................. 181
--only_stdout ............................................................................ 181
--output, -o .............................................................................. 181
--predef_macros ....................................................................... 182
--preinclude ............................................................................ 182
--preprocess ............................................................................. 182
--public_equ ............................................................................. 183
-r, --debug ............................................................................... 183
--remarks ............................................................................... 183
--require_prototypes ............................................................... 184
--ropi ....................................................................................... 184
--rwpi ....................................................................................... 185
--section ............................................................................... 185
--separate_cluster_for_initialized_variables ................................ 186
--silent ................................................................................... 186
--strict_ansi ........................................................................... 186
--thumb .................................................................................. 187
--use_unix_directory_separators ............................................. 187
--warnings_affect_exit_code .................................................... 187
--warnings_are_errors ............................................................ 187

Linker options ........................................................................... 189

Summary of linker options .......................................................... 189

Descriptions of options ............................................................... 191
--BE8 ....................................................................................... 191
--BE32 ..................................................................................... 191
--config ................................................................................... 192
--config_def ........................................................................... 192
--cpp_init_routine .................................................................. 193
--cpu ....................................................................................... 193
--define_symbol ...................................................................... 193
--diag_error ........................................................................... 194
--warnings_are_errors ................................................................. 206

Data representation ......................................................................... 209

Alignment ..................................................................................... 209
Alignment on the ARM core ......................................................... 209

Byte order ..................................................................................... 210

Basic data types ............................................................................ 210
Integer types ................................................................................ 210
Floating-point types ................................................................. 213

Pointer types ................................................................................ 215
Function pointers ......................................................................... 215
Data pointers ............................................................................... 215
Casting ......................................................................................... 215

Structure types ............................................................................. 216
Alignment ................................................................................... 216
General layout ............................................................................. 216
Packed structure types ............................................................... 217

Type qualifiers .............................................................................. 218
Declaring objects volatile ............................................................ 218
Declaring objects const ............................................................... 219

Data types in C++ ......................................................................... 219

Compiler extensions ..................................................................... 221

Compiler extensions overview ..................................................... 221
Enabling language extensions ...................................................... 222

C language extensions ................................................................ 222
Important language extensions .................................................... 222
Useful language extensions ......................................................... 225
Minor language extensions .......................................................... 228

Extended keywords ..................................................................... 233

General syntax rules for extended keywords ............................... 233
Type attributes ........................................................................... 233
Object attributes ......................................................................... 235

Summary of extended keywords ................................................ 236
## Descriptions of extended keywords

- `__absolute` ................................................................. 237
- `__arm` ............................................................................ 237
- `__big_endian` ............................................................... 237
- `__fiq` ............................................................................. 238
- `__interwork` ................................................................. 238
- `__intrinsic` ................................................................. 238
- `__irq` ............................................................................. 238
- `__little_endian` ............................................................ 239
- `__nested` ........................................................................ 239
- `__no_init` ....................................................................... 239
- `__noreturn` .................................................................... 240
- `__packed` ....................................................................... 240
- `__ramfunc` ..................................................................... 241
- `__root` ........................................................................... 241
- `__swi` .............................................................................. 242
- `__task` ............................................................................ 243
- `__thumb` ......................................................................... 243
- `__weak` .......................................................................... 244

## Pragma directives

- Summary of pragma directives ........................................ 245
- Descriptions of pragma directives ..................................... 246
  - `bitfields` ....................................................................... 246
  - `data_alignment` ............................................................ 247
  - `diag_default` ............................................................... 248
  - `diag_error` ................................................................... 248
  - `diag_remark` ............................................................... 248
  - `diag_suppress` ............................................................. 249
  - `diag_warning` ............................................................. 249
  - `include_alias` .............................................................. 249
  - `inline` ......................................................................... 250
  - `language` ..................................................................... 251
  - `location` ...................................................................... 251
message ................................................................. 252
object_attribute ...................................................... 252
optimize ................................................................. 252
pack ......................................................................... 253
__printf_args ......................................................... 254
required .................................................................... 254
rtmodel ................................................................. 255
__scanf_args .......................................................... 256
section ...................................................................... 256
swi_number ............................................................ 257
type_attribute ....................................................... 257
weak ......................................................................... 258

Intrinsic functions .................................................. 259

Summary of intrinsic functions ................................ 259

Descriptions of intrinsic functions ...................... 262
__CLZ ................................................................. 262
__disable_fiq ........................................................ 263
__disable_interrupt .............................................. 263
__disable_irq ........................................................ 263
__DMB ............................................................... 263
__DSB ............................................................... 263
__enable_fiq ........................................................ 264
__enable_interrupt .............................................. 264
__enable_irq ........................................................ 264
__get_BASEPRI .................................................. 264
__get_CONTROL ................................................ 264
__get_CPSR ....................................................... 265
__get_FAULTMASK .............................................. 265
__get_interrupt_state ......................................... 265
__get_LR ............................................................. 265
__get_PRIMASK .................................................. 266
__get_SB ............................................................. 266
__get_SP ............................................................. 266
__ISB ................................................................. 266
__LDC ................................................................. 266
__LDCL ............................................................... 266
__LDC2 ............................................................... 266
__LDC2L ............................................................. 266
__LDC_noidx ....................................................... 267
__LDCL_noidx ..................................................... 267
__LDC2_noidx ..................................................... 267
__LDC2L_noidx ................................................... 267
__LDREX ............................................................ 267
__MCR ............................................................... 267
__MRC ............................................................... 268
__no_operation .................................................. 268
__QADD ............................................................. 269
__QADD8 ........................................................... 269
__QADD16 .......................................................... 269
__QASX ............................................................. 269
__QDADD ........................................................... 269
__QDOUBL E ........................................................ 270
__QDSUB ........................................................... 270
__QFlag ............................................................. 270
__QSUB ............................................................. 270
__QSUB8 ........................................................... 271
__QSUB16 .......................................................... 271
__QSAX ............................................................. 271
__REV .............................................................. 271
__REVSH ........................................................... 271
__SADD8 ........................................................... 272
__SADD16 .......................................................... 272
__SASX ............................................................. 272
__SEL ............................................................... 272
__set_BASEPRI .................................................. 272
__set_CONTROL .................................................. 273
__set_CPSR ....................................................... 273
The linker configuration file ................................................................. 299

Overview .......................................................................................... 299

Defining memories and regions ............................................................ 300
  Define memory directive ................................................................. 300
  Define region directive ................................................................. 301

Regions .............................................................................................. 301
  Region literal .................................................................................. 301
  Region expression .......................................................................... 303
  Empty region ................................................................................. 304

Section handling ................................................................................ 304
  Define block directive .................................................................... 305
  Define overlay directive .................................................................. 306
  Initialize directive .......................................................................... 307
  Do not initialize directive ............................................................... 310
  Keep directive ............................................................................... 310
  Place at directive .......................................................................... 311
  Place in directive .......................................................................... 312

Section selection ............................................................................... 312
  Section-selectors .......................................................................... 313
  Extended-selectors ........................................................................ 315

Using symbols, expressions, and numbers ........................................... 316
  Define symbol directive ................................................................. 316
  Export directive ............................................................................. 317
  Expressions .................................................................................... 317
  Numbers ........................................................................................ 318

Structural configuration ...................................................................... 319
  If directive ..................................................................................... 319
  Include directive .......................................................................... 320

Section reference .............................................................................. 321

Summary of sections .......................................................................... 321

Descriptions of sections and blocks .................................................... 322
  .bss ............................................................................................ 322
  CSTACK ...................................................................................... 322
.cstart ........................................................................................................ 323
.data ........................................................................................................ 323
.data_init ............................................................................................... 323
.difunct ................................................................................................. 323
.HEAP ..................................................................................................... 323
.iar_dynexit .......................................................................................... 323
.intvec .................................................................................................. 324
 IRQ_STACK ........................................................................................... 324
.noinit .................................................................................................... 324
.rodata ................................................................................................... 324
.text ........................................................................................................ 324

IAR utilities .......................................................................................... 325

The IAR Archive Tool—iarchive .......................................................... 325
 Invocation syntax .................................................................................. 325
 Summary of iarchive commands .......................................................... 326
 Summary of iarchive options ................................................................. 327
 Descriptions of command line options ................................................. 327
 -f .......................................................................................................... 327
 --create ............................................................................................. 327
 --delete, -d ....................................................................................... 328
 --extract, -x ..................................................................................... 328
 -o ......................................................................................................... 329
 --replace, -r ..................................................................................... 329
 --silent, -S ......................................................................................... 329
 --toc, -t ............................................................................................. 330
 --symbols .......................................................................................... 330
 --verbose, -V .................................................................................... 330
 Diagnostic messages ........................................................................... 331

The IAR ELF Tool—ielftool ................................................................. 332
 Invocation syntax .................................................................................. 332
 Summary of ielftool options ................................................................. 333
 Descriptions of options ....................................................................... 333
 --bin ................................................................................................... 334

xxiii
The IAR ELF Dumper for ARM—ielfdumparm ........................................ 338
Invocation syntax .............................................................................. 339
Summary of ielfdumparm options ....................................................... 339
Descriptions of options .................................................................... 340
--all ................................................................................................... 340
-o, --output ....................................................................................... 340
--section, -s ....................................................................................... 340
--raw ................................................................................................ 341
The IAR ELF Object Tool—iobjmanip .............................................. 341
Invocation syntax .............................................................................. 341
Summary of iobjmanip options ........................................................ 342
Descriptions of command line options ............................................. 342
-f ....................................................................................................... 342
--remove_section ............................................................................ 343
--rename_section ............................................................................ 343
--rename_symbol ............................................................................ 343
--strip ................................................................................................ 344
Diagnostic messages ........................................................................ 344
The IAR Absolute Symbol Exporter—isymexport ......................... 346
Invocation syntax .............................................................................. 346
Summary of isymexport options ....................................................... 347
Descriptions of options .................................................................... 347
--edit ............................................................................................... 347
Steering files .................................................................................... 347
Contents

Show directive .................................................................................. 348
Hide directive .................................................................................. 348
Rename directive .............................................................................. 348
Diagnostic messages ........................................................................ 349

Implementation-defined behavior .......................................................... 353

Descriptions of implementation-defined behavior .......................... 353
Translation ....................................................................................... 353
Environment ..................................................................................... 354
Identifiers ......................................................................................... 354
Characters ......................................................................................... 354
Integers ............................................................................................. 354
Floating point ................................................................................... 356
Arrays and pointers .......................................................................... 357
Registers ........................................................................................... 357
Structures, unions, enumerations, and bitfields ............................... 357
Qualifiers .......................................................................................... 358
Declarators ....................................................................................... 358
Statements ........................................................................................ 358
Preprocessing directives .................................................................... 358
IAR DLIB Library functions ............................................................ 360

Index ...................................................................................................... 365
Tables

1: Typographic conventions used in this guide .................................................... xxxiii
2: Naming conventions used in this guide ............................................................ xxxiv
3: Command line options for specifying library and dependency files ............. 23
4: Sections holding initialized data ................................................................. 45
5: Description of a relocation error ................................................................. 57
6: Library configurations ..................................................................................... 61
7: Customizable items ....................................................................................... 64
8: Formatters for printf ...................................................................................... 66
9: Formatters for scanf ...................................................................................... 67
10: Descriptions of printf configuration symbols .............................................. 77
11: Descriptions of scanf configuration symbols ............................................... 78
12: Low-level I/O files ....................................................................................... 79
13: Functions with special meanings when linked with debug info .................... 84
14: Example of runtime model attributes ......................................................... 86
15: Registers used for passing parameters ......................................................... 98
16: Registers used for returning values ............................................................. 99
17: Call frame information resources defined in a names block ....................... 102
18: Exception stacks ......................................................................................... 114
19: Compiler optimization levels ....................................................................... 133
20: Compiler environment variables .................................................................. 146
21: ILINK environment variables .................................................................... 147
22: Error return codes ...................................................................................... 149
23: Compiler options summary ......................................................................... 156
24: Linker options summary ............................................................................. 189
25: Integer types .............................................................................................. 210
26: Floating-point types ................................................................................... 213
27: Section operators and their symbols ............................................................ 224
28: Extended keywords summary ..................................................................... 236
29:Pragma directives summary ......................................................................... 245
30: Intrinsic functions summary ..................................................................... 259
31: Predefined symbols .................................................................................... 284
32: Values for specifying different CPU core families in \_TID\_ ....................... 286
33: Traditional standard C header files—DLIB ................................................. 293
34: Embedded C++ header files ........................................................................ 294
35: Additional Embedded C++ header files—DLIB ........................................... 294
36: Standard template library header files .......................................................... 295
37: New standard C header files—DLIB ............................................................... 295
38: Examples of section selector specifications .................................................... 314
39: Section summary ........................................................................................... 321
40: iarchive parameters ....................................................................................... 326
41: iarchive commands summary ........................................................................ 326
42: iarchive options summary ............................................................................ 327
43: ielftool parameters ....................................................................................... 332
44: ielftool options summary ............................................................................. 333
45: ielfdumparm parameters ................................................................................ 339
46: ielfdumparm options summary ..................................................................... 339
47: iobjmanip parameters ................................................................................... 341
48: iobjmanip options summary ......................................................................... 342
49: ielftool parameters ....................................................................................... 346
50: isymexport options summary ....................................................................... 347
51: Message returned by strerror()—IAR DLIB library .................................... 363
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 IAR Embedded Workbench® IDE User Guide for ARM®. This guide contains a product overview, tutorials that can help you get started, conceptual and user information about the IDE and the IAR C-SPY® Debugger, and corresponding reference information.
What this guide contains

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

Part 1. Using the build tools

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

Part 2. Reference information

- External interface details provides reference information about how the compiler and linker interact with their environment—the invocation syntax, methods for passing options to the compiler and linker, environment variables, the include file search procedure, and the different types of compiler and linker output. The chapter also describes how the diagnostic system works.
● **Compiler options** explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.

● **Linker options** gives a summary of the options, and contains detailed reference information for each linker option.

● **Data representation** describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.

● **Compiler extensions** gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.

● **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.

● **IAR utilities** describes the IAR utilities that handle the ELF and DWARF object formats.

● **Implementation-defined behavior** describes how the compiler handles the implementation-defined areas of the C language standard.

---

**Other documentation**

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

● Using the IDE and the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide for ARM®*

● Programming for the IAR Assembler for ARM, refer to the *ARM® IAR Assembler Reference Guide*

● Using the IAR DLIB Library functions, refer to the online help system
Other documentation

- Porting application code and projects created with a previous IAR Embedded Workbench for ARM, refer to the ARM® IAR Embedded Workbench® Migration Guide.


All of these guides are delivered in hypertext PDF or HTML format on the installation media. Some of them are also delivered as printed books.

**FURTHER READING**

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

- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall. [The later editions describe the ANSI C standard.]
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]

We recommend that you visit these web sites:

- The Advanced RISC Machines Ltd web site, [www.arm.com](http://www.arm.com), contains information and news about the ARM cores, as well as information about the ARM Embedded Application Binary Interface (AEABI).
- The IAR Systems web site, [www.iar.com](http://www.iar.com), holds application notes and other product information.
- The web site [www.SevensAndNines.com](http://www.SevensAndNines.com), maintained by IAR Systems, provides an online user community and resource site for ARM developers.
Finally, the Embedded C++ Technical Committee web site, www.caravan.net/ec2plus, contains information about the Embedded C++ standard.

**Document conventions**

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

When referring to a directory in your product installation, for example arm\doc, the full path to the location is assumed, for example c:\Program Files\IAR Systems\Embedded Workbench 5.n\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><strong>computer</strong></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><strong>parameter</strong></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, [ 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><strong>bold</strong></td>
<td>Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.</td>
</tr>
<tr>
<td><strong>italic</strong></td>
<td>• A cross-reference within this guide or to another guide.</td>
</tr>
<tr>
<td></td>
<td>• Emphasis.</td>
</tr>
<tr>
<td>...</td>
<td>An ellipsis indicates that the previous item can be repeated an arbitrary number of times.</td>
</tr>
<tr>
<td>🍃</td>
<td>Identifies instructions specific to the IAR Embedded Workbench® IDE interface.</td>
</tr>
<tr>
<td>🚀</td>
<td>Identifies instructions specific to the command line interface.</td>
</tr>
</tbody>
</table>

*Table 1: Typographic conventions used in this guide*
Document conventions

**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 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++
- 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.

For a more detailed product overview, see the IAR Embedded Workbench® IDE User Guide for ARM®. There you can also read about the debugger.

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.

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 detailed information, see the ARM® IAR Assembler Reference Guide.

THE IAR ILINK LINKER

The IAR ILINK Linker 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

Because ILINK both uses and produces industry-standard ELF and DWARF as object format, additional IAR utilities that handle these formats can be used:

- 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 ARM ELF Dumper—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 IAR Embedded Workbench® IDE User Guide for ARM®.
IAR language overview

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

- C, the most widely used high-level programming language in the embedded systems industry. Using the IAR C/C++ Compiler for ARM, you can build freestanding applications that follow the standard ISO 9899:1990. This standard is commonly known as ANSI C.

- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. IAR Systems supports two levels of the C++ language:
  - 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++, 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 deviations from the standard. For more details, see the chapter Compiler extensions.

For information about how the compiler handles the implementation-defined areas of the C language, see the chapter Implementation-defined behavior.

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

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

Device support

To get a smooth start with your product development, the IAR product installation comes with 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 a vast amount of preconfigured files for supporting different devices.

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. To read more about these files, see the IAR Embedded Workbench® IDE User 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. There are examples 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 167 for additional information.

For detailed descriptions of 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 ISO/ANSI C, and are very useful when you want to make sure that the source code is portable.

For detailed descriptions of the pragma directives, see the chapter Pragma directives.

**Predefined Symbols**

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

For detailed descriptions of 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 detailed information, see Primitives for interrupts, concurrency, and OS-related programming, page 30.

**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 89.
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 memory types. 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 an overview see Controlling data and function placement in memory, page 129. The linker places sections of code in memory according to the directives you specify in the linker configuration file, see Placing code and data—the linker configuration file, page 42.
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, or SFRs. 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 5. For an example, see Accessing special function registers, page 140.

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 simply 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 31.

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 14.

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 121.

The ARM IAR build tools version 5.xx are not fully compatible with earlier versions of the product, see the *ARM® IAR Embedded Workbench® Migration Guide* for more information.

### 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 in the *IAR Embedded Workbench® IDE User Guide for ARM®*.

**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 relocatable assembler. Both produce
relocatable object files in the industry-standard format ELF, including the DWARF format for debug information.

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

This illustration shows the translation process:

![Diagram of 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:

![Linking Process Diagram]

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

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

For an in-depth description of the procedure performed by ILINK, see *The linking process*, page 41.
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 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 332.

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

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

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

Application execution—an overview

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

- Initialization phase
- Execution phase
- Termination phase.
THE INITIALIZATION PHASE

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

- **Hardware initialization**, which generally at least initializes the stack pointer.
  
  The hardware initialization is typically performed in the system startup code `cstartup.s` and if required, by an extra low-level routine that you provide. It might include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.

- **Software C/C++ system initialization**
  
  Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.

- **Application initialization**
  
  This depends entirely on your application. Typically, 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.

1. 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:

   ![Diagram of hardware initialization]

   Figure 4: Initializing hardware
Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

![Diagram of memory regions]

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:

```
3
```

4 Finally, the `main()` function is called:

```
4
```
For a detailed description about each stage, see *System startup and termination*, page 70. For more details about initialization of data, see *Initialization at system startup*, page 45.

**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.

To read more about this, see *System termination*, page 73.

---

**Basic project configuration**

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 **--entry** 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.
However, you must specify some additional options. 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 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 details about how to set options and for a list of all available options, see the chapters Compiler options, Linker options, and the IAR Embedded Workbench® IDE User 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 IAR Embedded Workbench® IDE User Guide for ARM® for information about setting the Processor variant option in the IDE.

Use the --cpu option to specify the ARM core; see --cpu, page 161 for syntax information.

CPU mode

The IAR C/C++ Compiler for ARM supports two CPU modes: ARM and Thumb. All functions and function pointers will compile in the mode that you specify, except those explicitly declared __arm or __thumb.

See the IAR Embedded Workbench® IDE User Guide for ARM® for information about setting the Processor variant or Chip option in the IDE.
Use the --arm or --thumb option to specify the CPU mode for your project; see --arm, page 160 and --thumb, page 187, for syntax information.

Interworking

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

See the IAR Embedded Workbench® IDE User 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 171, 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 IAR Embedded Workbench® IDE User Guide for ARM® for information about setting the FPU option in the IDE.

Use the --fpu option to use the coprocessor for floating-point operations; see --fpu, page 170, for syntax information.

Byte order

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

See the IAR Embedded Workbench® IDE User 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 168, for syntax information.

OPTIMIZATION FOR SPEED AND SIZE

The compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common subexpression elimination, static clustering, instruction scheduling, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

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

**RUNTIME ENVIRONMENT**

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

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

**Note:** Some tailoring might be required, for example to meet hardware requirements.

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

The way you set up a runtime environment and locate all the related files differs depending on which build interface you are using—the IDE or the command line.

**Setting up for the runtime environment in the IDE**

The library is automatically chosen by the linker according to the settings you made in *Project>Options>General Options*, on the pages *Library Configuration*, *Library Options*, and *Library Usage*.

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 61, for more information.
Based on which library configuration you choose and your other project settings, the correct library file is used automatically. For the device-specific include files, a correct include path is set up.

**Setting up for the runtime environment from the command line**

Use these command line options to explicitly specify the library and the dependency files:

<table>
<thead>
<tr>
<th>Command line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-I arm\inc</td>
<td>Specifies the include path to device-specific I/O definition files.</td>
</tr>
<tr>
<td>--dlib_config</td>
<td>Specifies the library configuration file, either DLib_Config_Normal.h or</td>
</tr>
<tr>
<td>C:...\configfile.h</td>
<td>DLib_Config_Full.h</td>
</tr>
</tbody>
</table>

Table 3: Command line options for specifying library and dependency files

Normally, it is not needed to specify a library file explicitly, as ILINK automatically uses the correct library file.

For a list of all prebuilt library object files for the IAR DLIB Library, see *Using a prebuilt library*, page 62. Here you also get information about how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

**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 65.
- The size of the stack and the heap, see *Setting up the stack*, page 52, and *Setting up the heap*, page 52, respectively.
Basic project configuration
Data storage

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

**Introduction**

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

**DIFFERENT WAYS TO STORE DATA**

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

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

- **Global variables 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 27.
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—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.

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

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

---

**Dynamic memory on the heap**

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

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

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

**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 the ISO/ANSI C standard, the compiler provides several extensions for writing functions in C. Using these, you can:

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

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

For more information about optimizations, see Efficient coding for embedded applications, page 125. For information about the available intrinsic functions for accessing hardware operations, see the chapter Intrinsic functions.

ARM and Thumb code

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

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

The size of all code pointers is 4 bytes. There are restrictions to implicit and explicit casts from code pointers to data pointers or integer types or vice versa. For further information about the restrictions, see Pointer types, page 215.
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, or in other words 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 70.

The keyword is specified before the return type:

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

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

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

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

```
const int myc[] = { 10, 20 };       // myc initializer in
// .rodata (ROM)
msg("Hello");                      // String literal in
// .rodata (ROM)
```

may be rewritten to:

```
static int myc[] = { 10, 20 };      // Initialized by cstartup
static char hello[] = "Hello";      // Initialized by cstartup
msg(hello);                         // hello stored in .data (RAM),
// initializers in .data_init
```

For more details, see *Initializing code—copying ROM to RAM*, page 54.

### 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`, `__fig`, `__swi`, and `__nested`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, and `__set_interrupt_state`.
Note: ARM Cortex-M has a different interrupt mechanism than other ARM devices, and for these devices a different set of primitives is available. For more details, see *Interrupts for ARM Cortex-M*, page 36.

**Interrupt Functions**

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 simply stops executing the code it runs, and starts executing an interrupt routine instead. It is extremely 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.

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.

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

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

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

**Installing Exception Functions**

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

The default implementation of the ARM exception vector table in the standard runtime library jumps to predefined functions that implement an infinite loop. Any exception that occurs for an event not handled by your application will therefore be caught in the infinite loop (B.).
The predefined functions are defined as weak symbols. A weak symbol is only included by the linker as long as no duplicate symbol is found. If another symbol is defined with the same name, it will take precedence. Your application can therefore simply define its own exception function by just defining it using the correct name.

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

- Undefined_Handler
- SWI_Handler
- Prefetch_Handler
- Abort_Handler
- IRQ_Handler
- FIQ_Handler

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

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

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

An interrupt function must have C linkage, read more in Calling convention, page 95.

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()
{
}
```

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 238, and `type_attribute`, page 257, 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 238, and type_attribute, page 257, 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;

    vector = VICVectAddr;       // Get interrupt vector.
    VICVectAddr = 0;            // Acknowledge interrupt in VIC.
    interrupt_task = (void(*)())vector;

    __enable_interrupt();       // Allow other IRQ interrupts
to be serviced from this
point.

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

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

SOFTWARE INTERRUPTS

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

Calling a software interrupt function

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

A __swi function can for example be declared like this:

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

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

Software interrupt functions follow the same calling convention regarding parameters and return values as an ordinary function, except for the stack usage, see Calling convention, page 95.

For more information, see __swi, page 242, and swi_number, page 257, 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 242.

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 setup 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 52.

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 141.

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types, with the restriction 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:

```cpp
class Device
{
    static __irq void handler();
};
```
Primitives for interrupts, concurrency, and OS-related programming
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 56.

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 debugger that supports ELF/DWARF, or it can be programmed into EPROM.

To handle ELF files, various tools are included. For a list of included utilities, see *Specific ELF tools*, page 4.
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 used device
- 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 51, and *Keeping symbols and sections*, page 51.

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 execute 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 details 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 complies with AEABI (ARM Embedded Application Binary Interface). 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 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 object file that contains the executable image and any debug information provided. This involves resolving symbolic references between sections, and locating relocatable values.
Optionally, produce a map file that lists the result of the section placement, the address of each global symbol, and finally, a summary of memory usage for each module and library.

This illustration shows the linking process:

Figure 8: The linking process

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

Note: To see the actual content of an ELF object file, use ielfdumparm. See The IAR ELF Dumper for ARM—ielfdumparm, page 338.

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};
initalize by copy { readwrite }; /* Initialize RW sections, exclude zero-initialized sections */
/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup};
/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in ROM: .rodata and .data_init */
place in RAM { readwrite, /* Place .data, .bss, and .noinit */
block STACK }; /* and STACK */

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

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

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

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

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

This illustration gives a schematic overview of how the application is placed in memory:
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 reference information about the linker configuration file, see the chapter *The linker configuration file*.

### Initialization at system startup

In ISO/ANSI 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><code>int i;</code></td>
<td>Read/write data, zero-init</td>
<td>.bss</td>
<td>None</td>
</tr>
<tr>
<td>Zero-initialized data</td>
<td><code>int i = 0;</code></td>
<td>Read/write data, zero-init</td>
<td>.bss</td>
<td>None</td>
</tr>
<tr>
<td>Initialized data (non-zero)</td>
<td><code>int i = 6;</code></td>
<td>Read/write data</td>
<td>.data</td>
<td>The initializer</td>
</tr>
<tr>
<td>Non-initialized data</td>
<td><code>__no_init int i;</code></td>
<td>Read/write data, zero-init</td>
<td>.noinit</td>
<td>None</td>
</tr>
<tr>
<td>Constants</td>
<td><code>const int i = 6;</code></td>
<td>Read-only data</td>
<td>.rodata</td>
<td>The constant</td>
</tr>
<tr>
<td>Code</td>
<td><code>__ramfunc void myfunc() {}</code></td>
<td>Read/write code</td>
<td>.text</td>
<td>The code</td>
</tr>
</tbody>
</table>

*Table 4: Sections holding initialized data*

**Note:** Clustering of static variables might group zero-initialized variables together with initialized data in `.data`. 
Initialization at system startup

For a summary of 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 in two sections, where the original initialized section will keep the name. The contents are placed in the new initializer section, which will keep 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 that the linker splits in .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 */
```

For detailed information and examples about how to configure the initialization, see Linking considerations, page 47.
Linking your application

This chapter lists aspects that you must consider when linking your application. This includes using ILINK options and tailoring the linker configuration file. Finally, this chapter provides some hints for troubleshooting.

Linking considerations

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

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

CHOOSING A LINKER CONFIGURATION FILE

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

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

These files contain the information required by ILINK. The only change 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.
Linking considerations

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

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

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

DEFINING YOUR OWN MEMORY AREAS

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

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

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

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

Each region definition must be tailored for the actual hardware.

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

Adding an additional region

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

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

Merging different areas into one region

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

/* Define the 2nd ROM region to have two areas. The first with the start address 0x80000 and 128 Kbytes large, and the 2nd with the start address 0xC0000 and 32 Kbytes large */
define region ROM2 = Mem:[from 0x80000 size 0x20000]  
     | Mem:[from 0xC0000 size 0x08000];

or equivalently

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

**Adding a region in a new memory**

To add a region in a new memory, write:

/* Define a 2nd addressable memory */  
define memory Mem2 with size = 64k;
/* Define a region for constants with start address 0 and 64 Kbytes large */  
define region CONSTANT = Mem2:[from 0 size 0x10000];

**Defining the unit size for a new memory**

If the new memory is not byte-oriented (8-bits per byte) you should define what unit size to use:

/* Define the bit addressable memory */  
define memory Bit with size = 256, unitbitsize = 1;

**PLACING SECTIONS**

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

/* Place sections with readonly content in the ROM region */  
place in ROM (readonly);
/* Place the constant symbols in the CONSTANT region */  
place in CONSTANT (readonly section .rodata);

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

For the result of each placement directive after linking, inspect the placement summary in the map file (the command line option --map).
Placing a section at a specific address in memory
To place a section at a specific address in memory, use the place at directive, for example:

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

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

```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
/* Places a variable in your own section MyOwnSection. */
const int MyVariable @ "MyOwnSection" = 5;
```

```c
name createSection
/* Create a section */
section myOwnSection:CONST
/* And fill it with constant bytes */
dcb 5, 6, 7, 8
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:

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:

/* Declares a section */
#pragma section = "TempStorage"

char *TempStorage()
{
  /* 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 325.

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 40.

**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 41.
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 196.

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{ };  
define block IRQ_STACK with size = 64, alignment = 8{ };  
```

Specify an appropriate size for your application.

To read more about the stack, see Stack considerations, page 113.

SETTING UP THE HEAP

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

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

Specify the appropriate size for your application.

SETTING UP THE ATEXIT LIMIT

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

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

CHANGING THE DEFAULT INITIALIZATION

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

- Choosing the packing algorithm
- Manual initialization
- Initializing code—copying ROM to RAM.
For information about the performed initializations, inspect the log file (the command line option --log initialization).

**Choosing packing algorithm**

To override the default packing algorithm, write for example:

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

To read more about the available packing algorithms, see *Initialize directive*, page 307.

**Manual initialization**

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

```plaintext
/* Sections MYOVERLAY1 and MYOVERLAY2 will be overlaid in MyOverlay */
define overlay MyOverlay ( section MYOVERLAY1 );
define overlay MyOverlay ( section MYOVERLAY2 );

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

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

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

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

```plaintext
#include <string.h>

/* Declare the sections. */

#pragma section = "MyOverlay"
#pragma section = "MyOverlay1InRom"
```
/* Function that switches in image 1 into the overlay. */

void SwitchToOverlay1()
{
    char *targetAddr = __section_begin("MyOverlay");
    char *sourceAddr = __section_begin("MyOverlay1InRom");
    char *sourceAddrEnd = __section_end("MyOverlay1InRom");
    int size = sourceAddrEnd - sourceAddr;
    memcpy(targetAddr, sourceAddrEnd, size);
}

Initializing code—copying ROM to RAM

Sometimes, an application copies pieces of code from flash/ROM to RAM. This can be easily achieved by ILINK for whole code regions. However, for individual functions, the __ramfunc keyword can be used, see Execution in RAM, page 30

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

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

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

/* Place both in a block */
define block RamCode { section RAMCODE };
define block RamCodeInit { section RAMCODE_init };

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

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

For more examples, see Interaction between the tools and your application, page 115.

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 };
```

To read more about the available compression algorithms, see Initialize directive, page 307.

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

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

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

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

### INTERACTION BETWEEN ILINK AND THE APPLICATION

ILINK provides the command line options --config_def and --define_symbol to define symbols which can be used for controlling the application. You can also use symbols to represent the start and end of a continuous memory area that is defined in the linker configuration file. For more details, see Interaction between the tools and your application, page 115.

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.
Hints for troubleshooting

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 115.

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 a prebuilt library, page 62.

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, use ielftool.

VENEERS

The ARM cores need to use veneers on two occasions:

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

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

For more information, see --no_veneers, page 202.

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 198
Linking your application

- The --map option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see --map, page 200.

**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

<table>
<thead>
<tr>
<th>Kind</th>
<th>R_XXX__YYV[0x1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>0x40000448</td>
</tr>
<tr>
<td></td>
<td>&quot;myfunc&quot; + 0x2c</td>
</tr>
<tr>
<td>Module</td>
<td>somecode.o</td>
</tr>
<tr>
<td>Section</td>
<td>7 (.text)</td>
</tr>
<tr>
<td>Offset</td>
<td>0x2c</td>
</tr>
<tr>
<td>Destination</td>
<td>0x90000000c</td>
</tr>
<tr>
<td></td>
<td>&quot;read&quot;</td>
</tr>
<tr>
<td>Module</td>
<td>read.o(iolib.a)</td>
</tr>
<tr>
<td>Section</td>
<td>6 (.text)</td>
</tr>
<tr>
<td>Offset</td>
<td>0x0</td>
</tr>
</tbody>
</table>

The message entries are described in this table:

<table>
<thead>
<tr>
<th>Message entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>The relocation directive that failed. The directive depends on the instruction used.</td>
</tr>
<tr>
<td>Location</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>
</tbody>
</table>

Table 5: Description of a relocation error
Hints for troubleshooting

Possible solutions

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

Possible solutions include ensuring that the two `.text` sections are allocated closer to each other or using some other calling mechanism that can reach the required distance. It is also possible that the referring function tried to refer to the wrong target and that this caused the range error.

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

<table>
<thead>
<tr>
<th>Message entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</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 <code>read.o</code> and the library <code>iolib.a</code>.</td>
</tr>
<tr>
<td></td>
<td>• The section number and section name. In this example, section number 6 with the name <code>.text</code>.</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 5: Description of a relocation error (Continued)
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 modify it—setting options, overriding default library modules, or building your own library—to optimize it for your application.

The chapter also covers system initialization and termination; how an application can control what happens before the function main is called, and how you can customize the initialization.

The chapter then describes how to configure functionality like locale and file I/O, how to get C-SPY® runtime support, and how to prevent incompatible modules from being linked together.

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. The IAR DLIB runtime environment can be used as is together with the debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:
● The runtime environment and its components
● Library selection.

For information about AEABI compliance, see AEABI compliance, page 121.

RUNTIME ENVIRONMENT FUNCTIONALITY

The runtime environment supports ISO/ANSI C and C++ including the standard template library. The runtime environment consists of the runtime library, which contains the functions defined by these standards, and include files that define the library interface.
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 register 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.
- Special compiler support for some functions, for instance functions for floating-point arithmetics.

The runtime environment support and the size of the heap must be tailored for the specific hardware and application requirements.

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

**LIBRARY SELECTION**

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.

IAR Embedded Workbench comes with a set of prebuilt runtime libraries. To get the required runtime environment, you can customize it by:

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: Normal or Full.

You can also make your own library configuration, but that requires that you *rebuild* the library. This allows you to get full control of the runtime environment.

**Note:** Your application project must be able to locate the library, include files, and the library configuration file. ILINK will automatically choose a prebuilt library suitable for the application.
SITUATIONS THAT REQUIRE LIBRARY BUILDING

Building a customized library is complex. Therefore, consider carefully whether it is really necessary.

You must build your own library when:

- There is no prebuilt library for the required combination of compiler options or hardware support
- You want to define your own library configuration with support for locale, file descriptors, multibyte characters, et cetera.

For information about how to build a customized library, see *Building and using a customized library*, page 68.

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 is.

These DLIB library configurations are available:

<table>
<thead>
<tr>
<th>Library configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal DLIB</td>
<td>No locale interface, C locale, no file descriptor support, no multibyte characters in <code>printf</code> and <code>scanf</code>, and no hexadecimal floating-point numbers in <code>strtod</code>.</td>
</tr>
<tr>
<td>Full DLIB</td>
<td>Full locale interface, C locale, file descriptor support, multibyte characters in <code>printf</code> and <code>scanf</code>, and hexadecimal floating-point numbers in <code>strtod</code>.</td>
</tr>
</tbody>
</table>

For further information, see *Building and using a customized library*, page 68.

The prebuilt libraries are based on the default configurations, see *Using a prebuilt library*, page 62. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.
LOW-LEVEL INTERFACE FOR DEBUG SUPPORT

If your application uses the DLIB low-level interface (see C-SPY runtime interface, page 84), you must implement support for the parts used by the application. However, if you must debug your application before this is implemented, you can temporarily use the semihosted debug support also provided as a library.

The low-level debugger runtime interface provided by DLIB is compatible with the semihosting interface provided by ARM Limited. The interface is implemented by a set of SVC (SuperVisor Call) instructions that generate exceptions from program control. The application invokes the appropriate semihosting call and the debugger then handles the exception. The debugger provides the required communication with the host computer.

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

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

Using a prebuilt library

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

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

In the IDE, the linker will include the correct library object file and library configuration file based on the options you select. See the IAR Embedded Workbench® IDE User Guide for ARM® for additional information.

If you build your application from the command line, you must specify the library configuration file for the compiler, either DLib_Config_Full.h or DLib_Config_Normal.h, for example:

```bash
--dlib_config C:\...\DLib_Config_Normal.h
```

You can find the library object files and the library configuration files in the subdirectory arm\lib, and the library configuration files in the arm\inc directory.
GROUPS OF LIBRARY FILES
The libraries are delivered in three groups of library functions:

- **C/C++ standard library functions**
  These are the functions defined by the ISO/ANSI C/C++ standard, for example functions like `printf` and `scanf`.

- **Runtime support functions**
  These are functions for system startup, initialization, floating-point arithmetics, ABI support, and some of the functions part of the ISO/ANSI C/C++ standard.

- **Debug support functions**
  These are functions for debug support for the semihosting interface.

Library filename syntax
The names of the libraries are constructed by the following constituents:

- `<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.
- `<cpu_mode>` is one of `t` or `a`, for Thumb and ARM, respectively.
- `<endian>` 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, software implementation compliant to AAPCS. It is `s` when the library is compiled with VFP support and compliant to AAPCS/STD. It is `v` when compiled with VFP support and using VFP registers in function calls; this is not AEABI-compliant. The supported version of VFP is `v1` when `architecture` is `4t` and `v2` when `architecture` is `5E` and higher.
- `<interworking>` is `i` when the library contains interworking code, otherwise it is `_`.
- `<library_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 205.
- `<ropi>` is `p` when the library contains read-only position-independent code, see `--ropi`, page 184.
- `<rwpi>` is `s` when the library contains read/write position-independent code, see `--rwpi`, page 185.
Library files for C/C++ standard library functions

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

\[ dl<architecture>_<cpu_mode><endian><fp_implementation><interworking><library_config><ropi><rwpi>.a \]

which more specifically means

\[ dl<4t|5E|6M|7M>_a|t><l|b><_|s|v><i>_n|f><p><n>.a \]

Library files for runtime support functions

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

\[ rt<architecture>_<cpu_mode><endian><fp_implementation>.a \]

which more specifically means

\[ rt<4t|5E|6M|7M>_a|t><l|b><_|s|v>.a \]

Library files for debug support functions

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

\[ sh<debug_interface>_<endian>.a \]

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, it is possible to customize parts of a library without rebuilding it. There are two different methods:

- Setting options for:
  - Formatters used by \texttt{printf} and \texttt{scanf}
  - The sizes of the heap and the stack
- Overriding library modules with your own customized versions.

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 65</td>
</tr>
<tr>
<td>Startup and termination code</td>
<td>System startup and termination, page 70</td>
</tr>
<tr>
<td>Low-level input and output</td>
<td>Standard streams for input and output, page 75</td>
</tr>
</tbody>
</table>

Table 7: Customizable items
Choosing formatters for printf and scanf

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, it is possible to optimize these functions even further, see Configuration symbols for printf and scanf, page 77.

### CHOOSING PRINTF FORMATTER

The printf function uses a formatter called _Printf. The default 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.

---

For a description about how to override library modules, see Overriding library modules, page 67.
Choosing formatters for printf and scanf

This table summarizes the capabilities of the different formatters:

<table>
<thead>
<tr>
<th>Formatting capabilities</th>
<th>_PrintfFull</th>
<th>_PrintfLarge</th>
<th>_PrintfSmall</th>
<th>_PrintfTiny</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>†</td>
<td>†</td>
<td>†</td>
<td>No</td>
</tr>
<tr>
<td>Floating-point specifiers a, and A</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Floating-point specifiers e, E, f, F, g, and G</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Conversion specifier n</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Format flag space, +, -, #, and 0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Length modifiers h, l, L, n, t, and z</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Field width and precision, including *</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>long long support</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

† Depends on the library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see Configuration symbols for printf and scanf, page 77.

Specifying the printf formatter in the IDE

To use any other formatter than the default (Full), choose Project>Options and select the General Options category. Select the appropriate option on the Library options page.

Specifying printf formatter from the command line

To use any other formatter than the default (_PrintfFull), add one of these ILINK command line options:

```
--redirect _Printfs_PrintfLarge
--redirect _Printfs_PrintfSmall
--redirect _Printfs=_PrintfTiny
```

CHOOSING SCANF FORMATTER

In a similar way to the printf function, scanf uses a common formatter, called _Scanf. The default version is very 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>_ScanfFull</th>
<th>_ScanfLarge</th>
<th>_ScanfSmall</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>††</td>
<td>††</td>
<td>††</td>
</tr>
<tr>
<td>Floating-point specifiers a, and A</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Floating-point specifiers e, E, f, F, g, and G</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Conversion specifier n</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Scan set [ and ]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment suppressing *</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>long long support</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 9: Formatters for scanf

† Depends on the library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see Configuration symbols for printf and scanf, page 77.

**Specifying scanf formatter in the IDE**

To use any other formatter than the default (Full), choose Project>Options and select the General Options category. Select the appropriate option on the Library options page.

**Specifying scanf formatter from the command line**

To use any other variant than the default (_ScanfFull), add one of these ILINK command line options:

```
--redirect _Scanf=_ScanfLarge
--redirect _Scanf=_PrintfSmall
```

**Overriding library modules**

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and cstartup. This can be done without rebuilding the entire library. This section describes the procedure for including your version of the module in the application project build process. The library files that you can override with your own versions are located in the arm\src\lib directory.

**Note:** If you override a default I/O library module with your own module, C-SPY support for the module is turned off. For example, if you replace the module __write with your own version, the C-SPY Terminal I/O window will not be supported.
Overriding library modules using the IDE

This procedure is applicable to any source file in the library, which means that
library_module.c in this example can be any module in the library.

1. Copy the appropriate library_module.c file to your project directory.
2. Make the required additions to the file (or create your own routine, using the default file as a model).
3. Add the customized file to your project.
4. Rebuild your project.

Overriding library modules from the command line

This procedure is applicable to any source file in the library, which means that
library_module.c in this example can be any module in the library.

1. Copy the appropriate library_module.c to your project directory.
2. Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same module name as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.
3. Compile the modified file using the same options as for the rest of the project:

   iccarm library_module.c

   This creates a replacement object module file named library_module.o.
4. Add library_module.o to the ILINK command line, either directly or by using an extended linker command file, for example:

   ilinkarm library_module.o

   Make sure that library_module.o is placed before the library on the command line. This ensures that your module is used instead of the one in the library.

   Run ILINK to rebuild your application.

   This will use your version of library_module.o, instead of the one in the library. For information about the ILINK options, see the chapter Linker options.

Building and using a customized library

In some situations, see Situations that require library building, page 61, it is necessary to rebuild the C/C++ standard library. In those cases you must:

- Set up a library project
Part 1. Using the build tools

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 has full library configuration, see Table 6, Library configurations, page 61.

In the IDE, modify the generic options in the created library project to suit your application, see Basic project configuration, page 19.

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 must have its own library configuration file based on either DLib_Config_Normal.h or DLib_Config_Full.h, which sets up that specific library with the required library configuration. For more information, see Table 7, Customizable items, page 64.

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

Modifying the library configuration file

In your library project, open the file DLib_Config_Normal.h or DLib_Config_Full.h, depending on your library, make a copy of the 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 Cortex-M, one of the following files is used instead of `cstartup.s`:

`thumb\cstartup_M.s` or `thumb\cstartup_M.c`

For information about how to customize the system startup code, see *Customizing system initialization*, page 74.

**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:

- 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 is not valid. 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:

- 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 details, see Initialization at system startup, page 45.
- Static C++ objects are constructed.
- The main function is called, which starts the application.

For an overview of the initialization phase, see Application execution—an overview, page 14.
SYSTEM TERMINATION

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

An application can terminate normally in two different ways:

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

As the ISO/ANSI 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 C 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.
Customizing system initialization

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

C-SPY interface to system termination

If your project is linked with the semihosted interface, the normal __exit and abort functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see C-SPY runtime interface, page 84.

Customizing system initialization

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

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

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

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

If you intend to rebuild the library, the source files are available in the template library project, see Building and using a customized library, page 68.

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

__LOW_LEVEL_INIT

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

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

As noted earlier, you should not modify the file cstartup.s if a customized version of __low_level_init is enough for your needs. However, if you do need to modify the file cstartup.s, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see Overriding library modules, page 67.

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 196.

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

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 primitive 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.

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 68. 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 C-SPY runtime interface, page 84.

Example of using __write

The code in this example uses memory-mapped I/O to write to an LCD display:

```
#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: A call to __write where buf has the value NULL is a command to flush the handle. 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:
#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 129.

### 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 65.

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:

<table>
<thead>
<tr>
<th>Printf configuration symbols</th>
<th>Includes support for</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DLIB_PRINTF_MULTIBYTE</td>
<td>Multibyte characters</td>
</tr>
<tr>
<td>_DLIB_PRINTF_LONG_LONG</td>
<td>Long long (ll qualifier)</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_FLOAT</td>
<td>Floating-point numbers</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_A</td>
<td>Hexadecimal floating-point numbers</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:

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must set up a library project, see Building and using a customized library, page 68. Define the configuration symbols according to your application requirements.

The library contains a large number of powerful functions for file I/O operations. If you use any of these functions, you must customize them to suit your hardware. To simplify adaptation to specific hardware, all I/O functions call a small set of primitive functions, each designed to accomplish one particular task; for example, `_open` opens a file, and `_write` outputs characters.

Note that file I/O capability in the library is only supported by libraries with full library configuration, see Library configurations, page 61. 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.

---

**printf configuration symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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, i, 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_AND_PRECISION</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 (Continued)**

**scanf configuration symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</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**

---

File input and output
Template code for these I/O files are 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 primitive 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 library with I/O debugging support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see Low-level interface for debug support, page 62.

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 during 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 68.

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

Building a library without support for locale interface

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

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

CHANGING LOCALES AT RUNTIME

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

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

`lang_REGION`

or

`lang_REGION.encoding`

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

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

Example

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

```
setlocale (LC_ALL, "sv_FI.UTF8");
```

Environment interaction

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

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

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

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

```
key=value\0
```

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

For example:

```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 67.

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 further information, see "Building and using a customized library", page 68.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For further information, see "Low-level interface for debug support", page 62.

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 67.
If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 68.

### Time

To make the `time` and `date` functions work, you must implement the three functions `clock`, `time`, and `__getzone`.

This does not require that you rebuild the library. You can find source templates in the files `clock.c` and `time.c`, and `getzone.c` in the `arm\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 67.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 68.

The default implementation of `__getzone` specifies UTC 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 further information, see *C-SPY runtime interface*, page 84.

### Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you must rebuild the library, see *Building and using a customized library*, page 68. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

### Assert

If you linked your application with support for runtime debugging, an assert will print a message on `stdout`. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. The `__ReportAssert` function generates the assert notification. You can find template code in the `arm\src\lib` directory. For further information, see *Building and using a customized library*, page 68.

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 287.
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 bytes. To change this limit, see *Setting up the atexit limit*, page 52.

C-SPY runtime interface

To include support for runtime and I/O debugging, you must link your application with the option *Semihosted* or *IAR breakpoint*, see *Low-level interface for debug support*, page 62.

In this case, special debugger variants of these library functions are linked to the application:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abort</code></td>
<td>C-SPY notifies that the application has called abort</td>
</tr>
<tr>
<td><code>clock</code></td>
<td>Returns the clock on the host computer</td>
</tr>
<tr>
<td><code>__close</code></td>
<td>C-SPY notifies that the end of the application was reached</td>
</tr>
<tr>
<td><code>__exit</code></td>
<td>Opens a file on the host computer</td>
</tr>
<tr>
<td><code>__read</code></td>
<td><code>stdin</code>, <code>stdout</code>, and <code>stderr</code> will be directed to the Terminal I/O window; all other files will read the associated host file</td>
</tr>
<tr>
<td><code>remove</code></td>
<td>Writes a message to the Debug Log window and returns -1</td>
</tr>
<tr>
<td><code>rename</code></td>
<td>Writes a message to the Debug Log window and returns -1</td>
</tr>
<tr>
<td><code>_ReportAssert</code></td>
<td>Handles failed asserts</td>
</tr>
<tr>
<td><code>__seek</code></td>
<td>Seeks in the associated host file on the host computer</td>
</tr>
<tr>
<td><code>system</code></td>
<td>Writes a message to the Debug Log window and returns -1</td>
</tr>
<tr>
<td><code>time</code></td>
<td>Returns the time on the host computer</td>
</tr>
<tr>
<td><code>__write</code></td>
<td><code>stdin</code>, <code>stdout</code>, and <code>stderr</code> will be directed to the Terminal I/O window, all other files will write to the associated host file</td>
</tr>
</tbody>
</table>

*Table 13: Functions with special meanings when linked with debug info*

**LOW-LEVEL DEBUGGER RUNTIME INTERFACE**

The low-level debugger runtime interface is used for communication between the application being debugged and the debugger itself. The debugger provides runtime services to the application via this 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 using 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 debugging purpose can be to produce debug trace printouts.

**THE DEBUGGER TERMINAL I/O WINDOW**

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Low-level interface for debug support*, page 62. 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.

See the IAR Embedded Workbench® IDE User Guide for ARM® for more information about the Terminal I/O window.

**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>Linker>Output and select the option Buffered terminal output in the IDE, or add this to the linker command line:

--redirect __write=__write_buffered

---

**Checking module consistency**

This section introduces the concept of runtime model attributes, a mechanism that you can use to ensure that modules are built using compatible settings.

When developing an application, it is important to ensure that incompatible modules are not used together. 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.
The tools provided by IAR Systems use a set of predefined runtime model attributes to automatically ensure module consistency.

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 uses several ways of checking them.

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>
<tr>
<td>file3</td>
<td>red</td>
<td>*</td>
</tr>
<tr>
<td>file4</td>
<td>red</td>
<td>spicy</td>
</tr>
<tr>
<td>file5</td>
<td>red</td>
<td>lean</td>
</tr>
</tbody>
</table>

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:

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

For detailed syntax information, see rtmodel, page 255.
You can also use the `rtmodel` assembler directive to specify runtime model attributes in your assembler source code. For example:

```assembly
rtmodel "color", "red"
```

For detailed syntax information, see the *ARM® IAR Assembler Reference Guide*.

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 detailed 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 issue is discussed in the section Calling assembler routines from C, page 92. The following two are covered in the section Calling convention, page 95.

The section Inline assembler, page 91, covers how to use inline assembler, but it also shows how data in memory is accessed.

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 101.

The recommended method for mixing C or C++ and assembler modules is described in Calling assembler routines from C, page 92, and Calling assembler routines from C++, page 94, respectively.

**INLINE ASSEMBLER**

It is possible to insert assembler code directly into a C or C++ function. The `asm` keyword inserts the supplied assembler statement in-line, see Inline assembler, page 225 for reference information. The following example demonstrates the use of the `asm` keyword. This example also shows the risks of using inline assembler.

```c
bool flag;

void foo()
{
    while (!flag)
    {
        asm("           ldr r2,[pc,#0]   
                   \n*   r2 = address of flag */
                   b .+8
                   \n*   jump over constant */
                   DCD flag
                   \n*   address of flag */
                   ldr r3,[pc,#0]   \n*   r3 = address of PIND */
                   b .+8
                   \n*   jump over constant */
                   DCD PIND
                   \n*   address of PIND */
                   ldr r0,[r3]   \n*   r0 = PIND */
                   str r0,[r2]*/;    \n*   flag = r0 */
    }
}
```

In this example, the assignment of `flag` is not noticed by the compiler, which means the surrounding code cannot be expected to rely on the inline assembler statement.

The inline assembler instruction will simply be inserted at the given location in the program flow. The consequences or side-effects 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.
Inline assembler sequences have no well-defined interface with the surrounding code generated from your C or C++ code. This makes the inline assembler code fragile, and will possibly also become a maintenance problem if you upgrade the compiler in the future. There are also several limitations to using inline assembler:

- The compiler’s various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Alignment cannot be controlled; this means, for example, that DC32 directives might be misaligned
- Auto variables cannot be accessed
- Alternative register names, mnemonics, and operators are not supported; read more about the -j assembler option in the ARM® IAR Assembler Reference Guide.

Inline assembler is therefore often best avoided. If no suitable intrinsic function is available, we recommend that you use modules written in assembler language instead of inline assembler, because the function call to an assembler routine normally causes less performance reduction.

### 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

```c
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:

```
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
Calling assembler routines from C++

The 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);
    }
};
```

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:

- Choosing a calling convention
- Function declarations
- C and C++ linkage
- Preserved versus scratch registers
- Function entrance
- Function exit
- Return address handling.
At the end of the section, some examples are shown to describe the calling convention in practice.

Unless otherwise noted, the calling convention used by the compiler adheres to AAPCS, a part of AEABI; see AEABI compliance, page 121.

**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"
{
    int F(int);
#endif
#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 94.

**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-bit) floating-point values</td>
<td>Passed using the first free register: R0–R3</td>
</tr>
<tr>
<td>long and double-precision (64-bit) values</td>
<td>Passed in first available register pair: R0:R1, or R2:R3</td>
</tr>
</tbody>
</table>

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.

![Stack parameters in memory](image)

The stack should be aligned to 8 at function entry.

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 register/register pair</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>R0</td>
</tr>
<tr>
<td>The memory address of a structure return value larger than 32 bits</td>
<td>R0</td>
</tr>
<tr>
<td><code>long long</code> and double-precision (64-bit) return values</td>
<td>R0:R1</td>
</tr>
</tbody>
</table>

Table 16: 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:

```
PUSH        {R4-R6,LR}      /* Function entry. */

add1:
ADDS       R0, R0, #0x1
BX          BX, LR
```

**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:

```
add1:
ADDS       R0, R0, #0x1
BX          BX, LR
```
Example 2
This example shows how structures are passed on the stack. Assume these declarations:

```c
struct MyStruct
{
    int a, b, c, d, 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 \( R0 \).

Call frame information

When you debug an application using C-SPY, you can view the call stack, that is, the chain of functions that called the current function. To make this possible, the compiler supplies debug information that describes the layout of the call frame, in particular information about where the return address is stored.

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the ARM® IAR Assembler Reference Guide.

CFI DIRECTIVES

The `CFI` directives provide C-SPY with information about the state of the calling function(s). Most important of this is the return address, and the value of the stack pointer at the entry of the function or assembler routine. Given this information, C-SPY can reconstruct the state for the calling function, and thereby unwind the stack.

A full description about the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice.

When describing the call frame information, the following three components must be present:

- A `names block` describing the available resources to be tracked
- A `common block` corresponding to the calling convention
A *data block* describing the changes that are performed on the call frame. This typically includes information about when the stack pointer is changed, and when permanent registers are stored or restored on the stack.

This table lists all the resources defined in the names block used by the compiler:

<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>S0–S31</td>
<td>Vector Floating Point (VFP) 32-bit coprocessor registers</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 17: 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:

   ```
   NAME cfiexample
   EXTERN F
   PUBLIC cfiExample
   ```
Assembler language interface

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)
CFI CFA R13+8
MOVS     R4,R0
MOVS     R0,R4
BL        P
ADDS     R0,R0,R4
POP       {R4,LR}
CFI R4 SameValue
CFI R14 SameValue
CFI CFA R13+0
Call frame information

BX     LR       ;; return
CFI EndBlock cfiBlock0

END

**Note:** The header file `cfiCommon.i` contains the macros `CFI_NAME_BLOCK`, `CFI_COMMON_ARM`, and `CFI_COMMON_Thumb` which declare a typical names block and two typical common blocks. These macros declare several resources, both concrete and virtual.
Using C++

IAR Systems supports two levels of the C++ language: The industry-standard Embedded C++ and IAR Extended Embedded C++. They are described in this chapter.

Overview

Embedded C++ is a subset of the C++ programming language 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.

STANDARD EMBEDDED C++

The following 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 late additions to the ISO/ANSI C++ standard. This is because they represent potential portability problems, due to that few development tools support the standard. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

Standard Embedded C++ lacks these features of C++:

- Templates
- Multiple and virtual inheritance
- Exception handling
Overview

IAR C/C++ Development Guide

Compiling and linking for ARM

• 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 `typeid`) 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.

ENABLING C++ SUPPORT

In the compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `-ec++` compiler option. See `-ec++`, page 167.
To take advantage of Extended Embedded C++ features in your source code, you must use the --eec++ compiler option. See --eec++, page 167.

To set the equivalent option in the IDE, choose Project>Options>C/C++ Compiler>Language.

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.

CLASSES

A class type class and struct in C++ can have static and non-static data members, and static and non-static function members. The non-static function members can be further divided into virtual function members, non-virtual function members, constructors, and destructors. For the static data members, static function members, and non-static non-virtual function members the same rules apply as for statically linked symbols outside of a class. In other words, they can have any applicable IAR-specific type and object attribute.

The non-static virtual function members can have any applicable IAR-specific type and object attribute as long as a pointer to the member function can be implicitly converted to the default function pointer type. The constructors, destructors, and non-static data members cannot have any IAR attributes.

The location operator @ can be used on static data members and on any type of function members.

For further information about attributes, see Type qualifiers, page 218.

Example

class MyClass
{
 public:
   // Locate a static variable 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 __thumb void th();

// Interworking assumed
virtual __arm void ah();

// 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
}
```

**TEMPLATES**

*Extended EC++* supports templates according to the C++ standard, except for the support of 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.
The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in Extended Embedded C++, page 106.

STL and the IAR C-SPY® Debugger

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.

To read more about displaying STL containers in the C-SPY debugger, see the IAR Embedded Workbench® IDE User Guide for ARM®.

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.

POINTER TO MEMBER FUNCTIONS

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a
When you use the compiler in 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
    friend class B; //According to 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.

**Example**

```cpp
class X
{
    public:
        __interwork void aF();
    
    void (__interwork X::*ap)() = &X::aF;
}
```
- In the declaration of a class member, a qualified name can be used, for example:
  ```
  struct A
  {
    int A::F(); // Possible when using IAR language extensions
    int G();    // According to standard
  };
  ```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:
  ```
  extern "C" void F(); // Function with C linkage
  void (*PF)()       // PF points to a function with C++ linkage
                    = &F; // Implicit conversion of function pointer.
  ```
  According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:
  ```
  bool X;
  char *P1 = X ? "abc" : "def"; // Possible when using IAR language extensions
  char const *P2 = X ? "abc" : "def"; // According to 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.

**Note:** If you use any of these constructions without first enabling language extensions, errors are issued.
C++ language extensions
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 332.

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 data section 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, CSTACK, 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 52, and Saving stack space and RAM memory, page 138.

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 95 and more specifically Special registers, page 97 and Stack parameters and layout, page 99.

EXCEPTION STACKS

The ARM architecture supports five exception modes which are entered when different exceptions occur. Each exception mode has its own stack to avoid corrupting the System/User mode stack.

The table shows proposed stack names for the various exception stacks, but any name can be used:

<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>Operating 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>
<tr>
<td>Abort</td>
<td>ABT_STACK</td>
<td>Stack for instruction fetch and data access memory abort interrupt handlers.</td>
</tr>
</tbody>
</table>

Table 18: 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 52.

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, et cetera.
Interaction between the tools and your application

- Creating an exported configuration symbol by using the command line option
  \texttt{--config\_def} or the configuration directive \texttt{define symbol}, and exporting the
  symbol using the \texttt{export symbol} directive. \texttt{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 \texttt{\_\_section\_begin}, \texttt{\_\_section\_end}, or
  \texttt{\_\_section\_size}, or the assembler operators \texttt{SFB}, \texttt{SFE}, or \texttt{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 \texttt{--entry} informs \texttt{ILINK} about the start label of the
  application. It is used by \texttt{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:

\begin{verbatim}
--define\_symbol NrOfElements=10
--config\_def HeapSize=1024
\end{verbatim}

The linker configuration file can look like this:

\begin{verbatim}
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 size defined by an ILINK option */
define block MyHEAP with size = HeapSize, alignment = 8 {};
place in RAM { block MyHEAP };
\end{verbatim}

Add these lines to your application source code:

\begin{verbatim}
#include <stdlib.h>
/* Use symbol defined by ILINK option to dynamically allocate
an array of elements with specified size */
extern char NrOfElements;
\end{verbatim}
typedef long Elements;
Elements *GetElementArray()
{
    return malloc(sizeof(Elements) * (int)& NrOfElements);
}

/* Use a symbol defined by ILINK option, a symbol that in the
   configuration file was made available to the application */
extern char HeapSize;

/* Declares the section that contains our heap */
#pragma section = "MyHEAP"

char *MyHeap()
{
    /* First get start of statically allocated section */
    char *p = __section_begin("MyHEAP");

    /* then we zero it, using the imported size */
    for (int i = 0; i < (int)& HeapSize; ++i)
    {
        p[i] = 0;
    }
    return p;
}

Checksum calculation

The IAR ELF Tool—ielftool—fills specific ranges of memory with a pattern and
then calculates a checksum for those ranges. The calculated checksum replaces the value
of an existing symbol in the input ELF image. The application can then verify that the
ranges did not change.

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

● Reserve a place, with an associated name and size, for the checksum calculated by
ielftool
● Choose a checksum algorithm, set up ielftool for it, and include source code for
the algorithm in your application
● Decide what memory ranges to verify and set up both ielftool and the source
code for it in your application source code.

Note: To set up ielftool in the IDE, choose Project>Options>Linker>Checksum.
CALCULATING A CHECKSUM

In this example, a checksum is calculated for ROM memory at 0x8002 up to 0x8FFF and the 2-byte calculated checksum is placed at 0x8000.

Creating a place for the calculated checksum

You can create a place for the calculated checksum in two ways; by creating a global C/C++ or assembler constant symbol with a proper size, residing in a specific section (in this example .checksum), or by using the linker option --place_holder.

For example, to create a 2-byte space for the symbol __checksum in the section .checksum, with alignment 4:

--place_holder __checksum,2,.checksum,4

To place the .checksum section, you must modify the linker configuration file. It can look like this (note the handling of the block CHECKSUM):

```
define memory Mem with size = 4G;
define region ROM_region = Mem:[from 0x8000 to 0x80000000 - 1];
define region RAM_region = Mem:[from 0x80000000 to 0x100000000 -2 ];
initialize by copy { rw };
do not initialize { section .noinit };

define block HEAP with alignment = 8, size = 16M {}
define block CSTACK with alignment = 8, size = 16K {}
define block IRQ_STACK with alignment = 8, size = 16K {}
define block FIQ_STACK with alignment = 8, size = 16K {}

define block CHECKSUM { ro section .checksum }
place at address Mem:0x0 { ro section .intvec }
place in ROM_region { ro, first block CHECKSUM }
place in RAM_region { rw, block HEAP, block CSTACK, block IRQ_STACK, block FIQ_STACK }
```

Running ielftool

To calculate the checksum, run ielftool:

```
ielftool --fill=0x00;0x8000–0x8FFF
--checksum=__checksum,2,crc16;0x8000–0x8FFF sourceFile.out
destinationFile.out
```

To calculate a checksum you also must define a fill operation. In this example, the fill pattern 0x0 is used. The checksum algorithm used is crc16.
Note that ielftool needs an unstripped input ELF image. If you use the --strip linker option, remove it and use the --strip ielftool option instead.

**ADDING A CHECKSUM FUNCTION TO YOUR SOURCE CODE**

To check the value of the ielftool generated checksum, it must be compared with a checksum that your application calculated. This means that you must add a function for checksum calculation (that uses the same algorithm as ielftool) to your application source code. Your application must also include a call to this function.

**A function for checksum calculation**

This function—a slow variant but with small memory footprint—uses the crc16 algorithm:

```c
unsigned short slow_crc16(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```
/* Start and end of the checksum range */
unsigned long ChecksumStart = 0x8000+2;
unsigned long ChecksumEnd   = 0xBFFF;
```c```
/* The checksum calculated by ielftool
   * (note that it lies 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 = slow_crc16(0,
                    (unsigned char *) ChecksumStart,
                    (ChecksumEnd - ChecksumStart+1));

    /* Rotate out the answer */
    calc = slow_crc16(calc, zeros, 2);

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

THINGS TO REMEMBER

When calculating a checksum, you must remember that:

● The checksum must be calculated from the lowest to the highest address for every
  memory range
● Each memory range must be verified in 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 a slow function 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 332.

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 less than four bytes, 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.

**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
- The IAR build tools do not support the use of exceptions and *rtti*
- 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.

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 122.

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 the IDE, use the `Project>Options>C/C++ Compiler>Extra Options` page to specify the `--aeabi` option.
On the command line, use the option `--aeabi` 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>
#ifndef _AEABI_PORTABLE
#error "header.h not AEABI compatible"
#endif
```
AEABI compliance
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 coprocessor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

For details 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. The format used by the `double` floating-point type depends on the setting of the `--double` compiler option.

Unless the application requires the extra precision that 64-bit floating-point numbers give, we recommend using 32-bit floating-point numbers instead.

By default, a *floating-point constant* in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, the `double` constant `1.0` is added and the result is converted back to a `float`:

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

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

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

For more information about floating-point types, see *Floating-point types*, page 213.
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 two reasons why this can be considered a problem:

- Due to 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 209.

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 further details about the `#pragma pack` directive, see `pack`, page 253.

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 167, for additional information.
Selecting data types

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;

/* Here the variables are used*/

void Test(void)
{
    IOPORT = 0;
    Way = 1;
    Out = 1;
}
```

This declares an I/O register byte IOPORT at address 0x1000. The I/O register has 2 bits declared, Way and Out. Note that both the inner structure and the outer union are anonymous.
Anonymous structures and unions are implemented in terms of objects named after the first field, with a prefix _A_ to place the name in the implementation part of the namespace. In this example, the anonymous union will be implemented through an object named _A_IOPORT.

**Controlling data and function placement in memory**

The compiler provides different mechanisms for controlling placement of functions and data objects in memory. To use memory efficiently, you should be familiar with these mechanisms to know which one is best suited for different situations. You can use:

- The @ operator and the #pragma location directive for absolute placement
  
  Use the @ operator or the #pragma location directive to place individual global and static variables at absolute addresses. The variables must be declared __no_init. This is useful for individual data objects that must be located at a fixed address, for example variables with external requirements. 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 --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. To read more about the --section option, see --section, page 185.

At compile time, data and functions are placed in different sections as described in Modules and sections, page 40. 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, as described in Placing code and data—the linker configuration file, page 42.
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. The variables must be declared using one of these combinations of keywords:

- __no_init
- __no_init and const (without initializers).

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: A variable placed in an absolute location should be defined in an include file, to be included in every module that uses the variable. An unused definition in a module will be ignored. A normal extern declaration—one that does not use an absolute placement directive—can refer to a variable at an absolute address; however, optimizations based on the knowledge of the absolute address cannot be performed.

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 @ 0x1000; /* OK */
```

This 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.

```c
#pragma location=0x1004
__no_init const int beta; /* OK */
```

The actual 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 @ 0x100C; /* Error, not __no_init */
__no_init int epsilon @ 0x1011; /* 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.

**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*. 

```c
volatile const __no_init int x @ 0x100;  /* Bad in C++ */
```
Examples of placing variables in named sections
In the following three examples, a data object is placed in a user-defined section.

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

#pragma location="CONSTANTS"
cnst int beta; /* OK */
```

Examples of placing functions in named sections
```c
void f(void) @ "FUNCTIONS";
void g(void) @ "FUNCTIONS"
{
}

#pragma location="FUNCTIONS"
void h(void);
```

## Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

### SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files.

For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to `optimize`, page 252, for information about the pragma directive.
Multi-file compilation units

In addition to applying different optimizations to different source files or even functions, you can also decide what a compilation unit consists of—one or several source code files.

By default, a compilation unit consists of one source file, but you can also use multi-file compilation to make several source files in a compilation unit. The advantage is that interprocedural optimizations such as inlining and cross jump have more source code to work on. Ideally, the whole application should be compiled as one compilation unit. However, for large applications this is not practical because of resource restrictions on the host computer. For more information, see --mfc, page 173.

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 166.

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:</td>
</tr>
<tr>
<td></td>
<td>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>
</tbody>
</table>

Table 19: Compiler optimization levels
Controlling compiler optimizations

Some of the performed optimizations can be individually enabled or disabled. For more information about these, see Fine-tuning enabled transformations, page 134.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option --no_inline, alternatively its equivalent in the IDE Function inlining, or the #pragma optimize directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Description</th>
</tr>
</thead>
</table>
| High (Maximum optimization)             | Same as above, and: Instruction scheduling  
|                                         | Cross jumping  
|                                         | Advanced register content analysis and optimization  
|                                         | Loop unrolling  
|                                         | Function inlining  
|                                         | Code motion  
|                                         | Type-based alias analysis  

Note: Some of the performed optimizations can be individually enabled or disabled.

Table 19: Compiler optimization levels (Continued)
● Static clustering
● Instruction scheduling.

**Common subexpression elimination**

Redundant re-evaluation of common subexpressions is by default eliminated at optimization levels **Medium** and **High**. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

**Note:** This option has no effect at optimization levels **None** and **Low**.

To read more about the command line option, see **--no_cse**, page 175.

**Loop unrolling**

It is possible to duplicate the loop body of a small loop, whose number of iterations can be determined at compile time, to reduce the loop overhead.

This optimization, 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. 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 read more about the command line option, see **--no_unroll**, page 179.

**Function inlining**

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level **High**, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler decides which functions to inline. 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 read more about the command line option, see **--no_inline**, page 176.

**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`.

### 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 ISO/ANSI standard-conforming C or C++ application code, this optimization can reduce code size and execution time. However, non-standard-conforming C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

**Note:** This option has no effect at optimization levels `None`, `Low`, and `Medium`.

To read more about the command line option, see `--no_tbaa`, page 178.

**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.
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.

Note: This option has no effect at optimization levels None, Low and Medium.

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. Also avoid taking the address of frequently accessed static variables.

● The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining might enable further optimizations. The compiler often inlines small functions declared static. The use of the #pragma inline directive and the C++
keyword inline gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. Too much inlining can decrease performance due to the limited number of registers. This feature can be disabled using the --no_inline command line option; see --no_inline, page 176.

- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more details, see Mixing C and assembler, page 89.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

FUNCTION PROTOTYPES

It is possible to declare and define functions using one of two different styles:

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. Using the prototyped style will also make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

To make the compiler verify that all functions have proper prototypes, use the compiler option Require prototypes (--require_prototypes).

Prototyped style

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

```c
int Test(char, int); /* Declaration */
int Test(char ch, int i) /* Definition */
{
    return i + ch;
}
```
Kernighan & Ritchie style

In K&R style—traditional pre-ISO/ANSI 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 0xFFFFFFFF. 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. To read more about the **volatile** type qualifier, see *Declaring objects volatile*, page 218.

### 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
/* system configuration register */
typedef struct {
  __REG32 se :1; /* stall enable, must be 0 */
  __REG32 ce :1; /* cache enable */
  __REG32 we :1;
  __REG32 cm :2; /* cache mode */
  __REG32 isbp :10; /* internal SRAM base pointer */
  __REG32 srbbp :10; /* special register bank base pointer */
  __REG32 :6;
} __syscfg_bits;

__IO_REG32_BIT(__SYSCFG,0x03FF0000,__READ_WRITE,__syscfg_bits);
```

By including the appropriate include file into the user code it is possible to access either the whole register or any individual bit (or bitfields) from C code as follows:

```c
/* whole register access */
__SYSCFG = 0x12345678;

/* Bitfield accesses */
__SYSCFG_bit.we = 1;
__SYSCFG_bit.cm = 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
#pragma diag_suppress=Pe940
#pragma optimize=no_inline
static unsigned long get_APSR( void )
{
    /* On function exit, 
       function return value should be present in R0 */
    asm( "MRS R0, APSR" );
}
#pragma diag_default=Pe940
#pragma optimize=no_inline
static void set_APSR( unsigned long value)
{
    /* On function entry, the first parameter is found in R0 */
    asm( "MSR APSR, R0" );
}
```

The general purpose register R0 is used for getting and setting the value of the special purpose register APSR. As the functions only contain inline assembler, the compiler will not interfere with the register usage. The register R0 is always used for return values. The first parameter is always passed in R0 if the type is 32 bits or smaller.

The same method can be used also for accessing other special purpose registers and specific instructions.

To read more about the risks of using inline assembler, see Inline assembler, page 91. For reference information about using inline assembler, see Inline assembler, page 225.

Note: Before you use inline assembler, see if you can use an intrinsic function instead. See Summary of intrinsic functions, page 259.

NON-INITIALIZED VARIABLES

Normally, the runtime environment will initialize all global and static variables when the application is started.

The compiler supports the declaration of variables that will not be initialized, using the __no_init type modifier. They can be specified either as a keyword or using the #pragma object_attribute directive. The compiler places such variables in a separate section.

For __no_init, the const keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a __no_init object an initial value.
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For information about the `__no_init` keyword, see page 239. Note that to use this keyword, language extensions must be enabled; see `-e`, page 167. For information about the `#pragma object_attribute`, see page 252.
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
- Compiler extensions
- Extended keywords
-Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- The linker configuration file
- Section reference
- IAR utilities
- Implementation-defined behavior.
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. Refer to the IAR Embedded Workbench® IDE User 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 that 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.
Invocation syntax

For example, when linking the object file `prog.o`, use this command:

```
ilinxarm 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 `ilinxarm` commands; see Invocation syntax, page 145.

- Via environment variables
  
  The compiler and linker automatically append the value of the environment variables to every command line; see Environment variables, page 146.

- Via a text file, using the `-f` option; see `-f`, page 169.

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:</td>
</tr>
<tr>
<td></td>
<td><code>C_INCLUDE=c:\program files\iar systems\embedded workbench \arm\inc;c:\headers</code></td>
</tr>
<tr>
<td>QCCARM</td>
<td>Specifies command line options; for example: <code>QCCARM=-lA asm.lst</code></td>
</tr>
</tbody>
</table>

Table 20: 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:</td>
</tr>
<tr>
<td></td>
<td>ILINKARM_CMD_LINE=--config full.icf</td>
</tr>
<tr>
<td></td>
<td>--silent</td>
</tr>
</tbody>
</table>

| Table 21: 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, 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 171.
2. The directories specified using the `C_INCLUDE` environment variable, if any, see Environment variables, page 146.

- 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
crc 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:

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

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

```plaintext
dir\include
```

Current file is `src.h`. 
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 172. 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. To read more about diagnostic messages, see Diagnostics, page 150.

- 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 149.

- 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 <code>--warnings_affect_exit_code</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 22: Error return codes

ILINK output

ILINK can produce the following output:

- An absolute executable image
  The final output produced by the IAR ILINK Linker is an absolute object file containing the executable image that can be put into an EPROM, downloaded to a hardware emulator, or executed on your PC using the IAR C-SPY Debugger Simulator. By default, the file has the filename extension `.out`. The output format is always in ELF, which optionally includes debug information in the DWARF format.

- Optional logging information
  During operation, ILINK logs its decisions on `stdout`, and optionally to a file. For example, if a library is searched, whether a required symbol is found in a library module, or whether a module will be part of the output. Timing information for each ILINK subsystem is also logged.

- Optional map files
  A linker map file—containing summaries of linkage, runtime attributes, memory, and placement, as well as an entry list—can be generated by the ILINK option `--map`, see `--map`, page 200. 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. To read more about diagnostic messages, see `Diagnostics`, page 150.
Diagnostics

- Error return codes
  ILINK returns status information to the operating system which can be tested in a batch file, see Error return codes, page 149.
- 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 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.

MESSAGE FORMAT FOR THE LINKER

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from ILINK is produced in the form:

```
level[tag]: message
```
with these elements:

- **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 map file.

Use the option `--diagnostics_tables` to list all possible linker diagnostic messages.

### SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

- **Remark**
  A diagnostic message that is produced when the compiler or linker finds a construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 183.

- **Warning**
  A diagnostic message that is produced when the compiler or linker finds a problem which is of concern, but not so severe as to prevent the completion of compilation or linking. Warnings can be disabled by use of the command line option `--no_warnings`, see page 180.

- **Error**
  A diagnostic message that is produced when the compiler or linker finds a serious error. An error will produce a non-zero exit code.

- **Fatal error**
  A diagnostic message that is produced when the compiler finds a condition that not only prevents code generation, but which makes further processing pointless. After the message is issued, compilation terminates. A fatal error will produce a non-zero exit code.

### SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all diagnostics messages, except for fatal errors and some of the regular errors.
See *Summary of compiler options*, page 156, for a description of the compiler options that are available for setting severity levels.

For the compiler see also the chapter *Pragma directives*, for a description of 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.

Refer to the *IAR Embedded Workbench® IDE User 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 146.

**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:

```
--mrisac2004=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\n
```

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=MyDiagnosics.lst
or
--diagnostics_tables MyDiagnosics.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:

```
-I A 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 also take a path. The path can be relative or absolute. For example, to generate a listing to the file `List.lst` in the directory `..\listings\`:
  
  ```
  iccarm prog.c -l ..\listings\List.lst
  ```
For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name was specified with the option `-o`, in which case that name is used. For example:

```
iccarm prog.c -l ..\listings\``

The produced list file will have the default name `..\listings\prog.lst`

- The current directory is specified with a period (`.`). For example:

```
iccarm prog.c -l .```

- `/` can be used instead of `\` as the directory delimiter.

- By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

```
iccarm prog.c -l -```

**Additional rules**

These rules also apply:

- When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead, you can prefix the parameter with two dashes; this example will create a list file called `-r`:

```
iccarm prog.c -l --r```

- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

```
--diag_warning=Be0001,Be0002
```

Alternatively, the option can be repeated for each argument, for example:

```
--diag_warning=Be0001
--diag_warning=Be0002
```
## Summary of compiler options

This table summarizes the compiler command line options:

<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 functions</td>
</tr>
<tr>
<td>--arm</td>
<td>Sets the default function mode to ARM</td>
</tr>
<tr>
<td>--char_is_signed</td>
<td>Treats char as signed</td>
</tr>
<tr>
<td>--cpu</td>
<td>Specifies a processor variant</td>
</tr>
<tr>
<td>--cpu_mode</td>
<td>Selects the default mode for functions</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>Determines the library configuration file</td>
</tr>
<tr>
<td>-e</td>
<td>Enables language extensions</td>
</tr>
<tr>
<td>--ec++</td>
<td>Enables Embedded C++ syntax</td>
</tr>
<tr>
<td>--eec++</td>
<td>Enables Extended Embedded C++ syntax</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>
</tbody>
</table>

*Table 23: Compiler options summary*
<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--fpu</td>
<td>Selects the type of floating-point unit</td>
</tr>
<tr>
<td>--header_context</td>
<td>Lists all referred source files and header files</td>
</tr>
<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>--mfc</td>
<td>Enables multi file compilation</td>
</tr>
<tr>
<td>--migration_preprocessor_extensions</td>
<td>Extends the preprocessor</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_const_align</td>
<td>Disables the alignment optimization for constants.</td>
</tr>
<tr>
<td>--no_cse</td>
<td>Disables common subexpression elimination</td>
</tr>
<tr>
<td>--no_fragments</td>
<td>Disables section fragment handling</td>
</tr>
<tr>
<td>--no_guard_calls</td>
<td>Disables guard calls for static initializers</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_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>
<tr>
<td>--no_tbaa</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>
</tbody>
</table>

Table 23: Compiler options summary (Continued)
<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--no_unroll</code></td>
<td>Disables loop unrolling</td>
</tr>
<tr>
<td><code>--no_warnings</code></td>
<td>Disables all warnings</td>
</tr>
<tr>
<td><code>--no_wrap_diagnostics</code></td>
<td>Disables wrapping of diagnostic messages</td>
</tr>
<tr>
<td><code>-O</code></td>
<td>Sets the optimization level</td>
</tr>
<tr>
<td><code>-o</code></td>
<td>Sets the object filename</td>
</tr>
<tr>
<td><code>--only_stdout</code></td>
<td>Uses standard output only</td>
</tr>
<tr>
<td><code>--output</code></td>
<td>Sets the object filename</td>
</tr>
<tr>
<td><code>--predef_macros</code></td>
<td>Lists the predefined symbols.</td>
</tr>
<tr>
<td><code>--preinclude</code></td>
<td>Includes an include file before reading the source file</td>
</tr>
<tr>
<td><code>--preprocess</code></td>
<td>Generates preprocessor output</td>
</tr>
<tr>
<td><code>--public_equ</code></td>
<td>Defines a global named assembler label</td>
</tr>
<tr>
<td><code>-r</code></td>
<td>Generates debug information</td>
</tr>
<tr>
<td><code>--remarks</code></td>
<td>Enables remarks</td>
</tr>
<tr>
<td><code>--require_prototypes</code></td>
<td>Verifies that functions are declared before they are defined</td>
</tr>
<tr>
<td><code>--ropi</code></td>
<td>Generates code that uses PC-relative references to address code and read-only data.</td>
</tr>
<tr>
<td><code>--rwpi</code></td>
<td>Generates code that uses an offset from the static base register to address-writable data.</td>
</tr>
<tr>
<td><code>--section</code></td>
<td>Changes a section name</td>
</tr>
<tr>
<td><code>--separate_cluster_for_initialized_variables</code></td>
<td>Separates initialized and non-initialized variables</td>
</tr>
<tr>
<td><code>--silent</code></td>
<td>Sets silent operation</td>
</tr>
<tr>
<td><code>--strict_ansi</code></td>
<td>Checks for strict compliance with ISO/ANSI C</td>
</tr>
<tr>
<td><code>--thumb</code></td>
<td>Sets default function mode to Thumb</td>
</tr>
<tr>
<td><code>--use_unix_directory_separators</code></td>
<td>Uses / as directory separator in paths</td>
</tr>
<tr>
<td><code>--warnings_affect_exit_code</code></td>
<td>Warnings affects exit code</td>
</tr>
<tr>
<td><code>--warnings_are_errors</code></td>
<td>Warnings are treated as errors</td>
</tr>
</tbody>
</table>

Table 23: Compiler options summary (Continued)
Descriptions of options

The following section gives detailed reference information about each compiler option.

Note that if you use the options page Extra Options to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--aapcs

Syntax

--aapcs={std|vfp}

Parameters

std

Processor registers are used for floating-point parameters and return values in function calls according to standard AAPCS. std is the default when the --aeabi compiler option is used or the software FPU is selected. Note that this calling convention enables guard calls.

vfp

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 is selected and --aeabi is not used.

Description

Use this option to specify the calling convention.

Project>Options>C/C++ Compiler>Extra Options.

--aeabi

Syntax

--aeabi

Description

Use this option to generate AEABI-compliant object code.

See also

AEABI compliance, page 121 and --no_guard_calls, page 176.

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.

See also: `__irq`, page 238.

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: `--interwork`, page 171 and `__interwork`, page 238.

Project>Options>General Options>Target>Processor mode>Arm

**--char_is_signed**

Syntax: `--char_is_signed`

Description: By default, the compiler interprets the `char` type as unsigned. Use this option to make the compiler interpret the `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. If you use this option, you might get type mismatch warnings from the linker, because the library uses `unsigned char`.

Project>Options>C/C++ Compiler>Language>Plain ‘char’ is
--cpu

Syntax
--cpu=core

Parameters

<table>
<thead>
<tr>
<th>Core</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM7TDMI</td>
<td>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:</td>
</tr>
<tr>
<td>ARM946E-S</td>
<td>Cortex-M0</td>
</tr>
<tr>
<td>ARM1176JF</td>
<td>Cortex-M1</td>
</tr>
<tr>
<td>ARM1176JF-S</td>
<td>Cortex-M1*</td>
</tr>
<tr>
<td>ARM710T</td>
<td>Cortex-M3</td>
</tr>
<tr>
<td>ARM720T</td>
<td>Cortex-M4</td>
</tr>
<tr>
<td>ARM740T</td>
<td>Cortex-M4F</td>
</tr>
<tr>
<td>ARM7EJ-S</td>
<td>XScale</td>
</tr>
<tr>
<td>ARM926EJ-S</td>
<td>XScale-IR7</td>
</tr>
<tr>
<td>ARM9TDMI</td>
<td>Cortex-M0</td>
</tr>
<tr>
<td>ARM922T</td>
<td>Cortex-M1</td>
</tr>
<tr>
<td>ARM926EJ-S</td>
<td>Cortex-M1*</td>
</tr>
<tr>
<td>ARM920T</td>
<td>Cortex-M4</td>
</tr>
<tr>
<td>ARM940T</td>
<td>Cortex-R4</td>
</tr>
<tr>
<td>ARM9E</td>
<td>Cortex-R4F</td>
</tr>
<tr>
<td>ARM9E-S</td>
<td>XScale</td>
</tr>
<tr>
<td>ARM9E</td>
<td>Cortex-M4F</td>
</tr>
<tr>
<td>ARM9E</td>
<td>XScale-IR7</td>
</tr>
<tr>
<td>ARM9E</td>
<td>(alias for XScale)</td>
</tr>
<tr>
<td>ARM9E-S</td>
<td>(alias for XScale-IR7)</td>
</tr>
</tbody>
</table>

See also

Processor variant, page 20.
--cpu_mode

**Syntax**

```text
--cpu_mode={arm|a|thumb|t}
```

**Parameters**

- `arm, a (default)`: Selects the arm mode as the default mode for functions
- `thumb, t`: Selects the thumb mode as the default mode for functions

**Description**

Use this option to select the default mode for functions. This setting must be the same for all files included in a program, unless they are interworking.

**See also**

--interwork, page 171 and __interwork, page 238.

-D

**Syntax**

```text
-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:

- `-Dsymbol` is equivalent to:
  ```
  #define symbol 1
  ```

To get the equivalence of:

- `#define FOO` specify the `=` sign but nothing after, for example:
  ```
  -DFOO=
  ```
---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.

---dependencies

**Syntax**

`--dependencies[=#l|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 154.

**Description**

Use this option to make the compiler list all source and header files opened by the compilation into a file with the default filename extension `i`.

**Example**

If `--dependencies` or `--dependencies=i` is used, the name of each opened source file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output uses makefile style. For each source 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 a source file. For example:

```
foo.o: c:\iar\product\include\stdio.h
foo.o: d:\myproject\include\foo.h
```

An example of using `--dependencies` with a popular make utility, such as gmake (GNU make):

```
$(IC) $(ICFLAGS) $< --dependencies=m $*.d
```

---
Descriptions of options

That is, in addition to producing an object file, the command also produces a dependency file in makefile style (in this example, using the extension `.d`).

2 Include all the dependency files in the makefile using, for example:

   -include $(sources:.c=.d)

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

This option is not available in the IDE.

--diag_error

Syntax

   --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 violation of the C or C++ language rules, of such severity that object code will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors

--diag_remark

Syntax

   --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 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.

Project>Options>C/C++ Compiler>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>C/C++ Compiler>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 compiler to stop before compilation is completed. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>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 154.

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 from the compilation unit. This enhances interprocedural optimizations such as inlining, cross call, and cross jump by limiting their scope to public functions and variables that are actually used.

This option is only useful when all source files are compiled as one unit, which means that the **--mfc** compiler option is used.

**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 173 and Multi-file compilation units, page 133.
```

**Project>Options>C/C++ Compiler>Discard unused publics**

**--dlib_config**

**Syntax**

```
--dlib_config filename
```

**Parameters**

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 154.

**Description**

Each runtime library has a corresponding library configuration file. Use this option to specify the library configuration file for the compiler. Make sure that you specify a configuration file that corresponds to the library you are using.

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 a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 62.

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 68.

To set related options, choose:

**Project>Options>General Options>Library Configuration**
Compiler options

- `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.

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

**See also**

*The chapter Compiler extensions.*

`--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>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++.

*Project>Options>C/C++ Compiler>Language>Extended Embedded C++*

Extended Embedded C++, page 106.
--enable_hardware_workaround

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

Parameters
waid

Description
Use this option to make the compiler generate a workaround for a specific hardware problem.

See also
The release notes 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>Enable multibyte support

--endian

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

Parameters
big, b
little, l (default)

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.
--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 211.

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 154.
Descriptions

Use this option to make the compiler read command line options from the named file, with the default filename extension .xcl.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

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

--fpu

Syntax

--fpu={VFPv1|VFPv2|VFPv3|VFPv3_d16|VFPv4_sp|VFP9-S|none}

Parameters

- **VFPv1**
  For a vector floating-point unit conforming to the architecture VFPv1.

- **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_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 21.

Project>Options>General Options>Target>FPU

IAR C/C++ Development Guide

Compiling and linking for ARM
--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.

See also
Include file search procedure, page 147.

Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

--interwork

Syntax
--interwork

Description
Use this option to generate interworking code.

In code compiled with this option, functions will by default be of the type interwork. It
will be possible to mix files compiled as arm and thumb (using the --cpu_mode option)
as long as they are all compiled with the --interwork option.

Note: Source code compiled for an ARM architecture v5 or higher, or AEABI
compliance is interworking by default.

Project>Options>General Options>Target>Generate interwork code
-l

Syntax

-l[a|A|b|B|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
D C or C++ list file with assembler source as comments, but without
instruction offsets and hexadecimal byte values
N No diagnostics in file
H Include source lines from header files in output. Without this
option, only source lines from the primary source file are included

* This makes the list file less useful as input to the assembler, but more useful for reading by a
human.

For information about specifying a filename or a directory, see Rules for specifying a
filename or directory as parameters, page 154.

Description

Use this option to generate an assembler or C/C++ listing to a file. Note that this option
can be used one or more times on the command line.

To set related options, choose:

Project>Options>C/C++ Compiler>List
### --legacy

**Syntax**

`--legacy={RVCT3.0}`

**Parameters**

<table>
<thead>
<tr>
<th>RVCT3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generates object code linkable with the linker in RVCT3.0. Use this mode together with the <code>--aeabi</code> option to export code that should be linked with the linker in RVCT3.0.</td>
</tr>
</tbody>
</table>

**Description**

Use this option to generate code compatible with older tool chains.

See also

- `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 makes interprocedural optimizations such as inlining, cross call, and cross jump possible.

**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**

- `--discard_unused_publics`, page 166, `-o`, `--output`, page 181, and `Multi-file compilation units`, page 133.

See also

- `Project>Options>C/C++ Compiler>Multi-file compilation`

### --migration_preprocessor_extensions

**Syntax**

`--migration_preprocessor_extensions`

**Description**

If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you might want to use this option. Use this option to use the following in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
Descriptions of options

- All symbol names (including typedefs and variables).

**Note:** If you use this option, not only will the compiler accept code that does not conform to the ISO/ANSI C standard, but it will also reject some code that *does* conform to the standard.

**Important!** Do not depend on these extensions in newly written code, because support for them might be removed in future compiler versions.

**Project>Options>C/C++ Compiler>Language>Enable IAR migration preprocessor extensions**

**--no_clustering**

**Syntax**

--no_clustering

**Description**

Use this option to disable static clustering optimizations. When static clustering is enabled, static and global variables 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. These optimizations, which are performed at optimization levels Medium and High, normally reduce code size and execution time.

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

**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Static clustering**

**--no_code_motion**

**Syntax**

--no_code_motion

**Description**

Use this option to disable code motion optimizations. These optimizations, which are performed at the optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code might be difficult to debug.

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

**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion**
--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 209.

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

--no_cse

Syntax: `--no_cse`

Description: Use this option to disable common subexpression elimination. At the optimization levels Medium and High, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

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

To set this option, use Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination.

--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 51.

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

--no_guard_calls

Syntax

--no_guard_calls

Description

If the --aeabi compiler option is used, the compiler produces extra library calls that guard the initialization of static variables in file scope. These library calls are only meaningful in an OS environment where you must make sure that these variables are not initialized by another concurrent process at the same time.

Use this option to remove these library calls.

*Note:* To be AEABI compliant, this option must not be used.

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

--no_inline

Syntax

--no_inline

Description

Use this option to disable function inlining. Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level High, normally reduces execution time and increases code size. The resulting code might also be difficult to debug.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed than for size.

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

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--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__.
Part 2. Reference information

Compiler options

See also

Descriptions of predefined preprocessor symbols, page 284.

This option is not available in the IDE.

--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 184 and --rwpi, page 185.

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. 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. This optimization, which is performed at optimization level High, normally reduce execution time. However, the resulting code might be difficult to debug.

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

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling
--no_tbaa

Syntax
--no_tbaa

Description
Use this option to disable type-based alias analysis. When this options is not used, the
compiler is free to assume that objects are only accessed through the declared type or
through unsigned char.

See also
Type-based alias analysis, page 136.

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, may 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: `--interwork`, page 171 and `__interwork`, page 238.

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. The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities.

This optimization, which is 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. Different heuristics are used when optimizing for speed and size.

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

To set this option, use `Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling`. 
### --no_warnings

**Syntax**

```
--no_warnings
```

**Description**

By default, the compiler issues warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

### --no_wrap_diagnostics

**Syntax**

```
--no_wrap_diagnostics
```

**Description**

By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

### -O

**Syntax**

```
-O[n|l|m|h|hs|hz]
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>None* (Best debug support)</td>
</tr>
<tr>
<td>l (default)</td>
<td>Low*</td>
</tr>
<tr>
<td>m</td>
<td>Medium</td>
</tr>
<tr>
<td>h</td>
<td>High, balanced</td>
</tr>
<tr>
<td>hs</td>
<td>High, favoring speed</td>
</tr>
<tr>
<td>hz</td>
<td>High, favoring size</td>
</tr>
</tbody>
</table>

*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 132.

Project>Options>C/C++ Compiler>Optimizations

---

**-o, --output**

**Syntax**

```bash
-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 154.

**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.

---

**--only_stdout**

**Syntax**

```bash
--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**

```bash
--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 154.

**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 154.

Description
Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the predef filename extension.

Note that this option requires that you specify a source file on the command line.

This option is not available in the IDE.

--preinclude

Syntax
--preinclude includefile

Parameters
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 154.

Description
Use this option to make the compiler include 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Preserve comments</td>
</tr>
<tr>
<td>n</td>
<td>Preprocess only</td>
</tr>
<tr>
<td>l</td>
<td>Generate #line directives</td>
</tr>
</tbody>
</table>

For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 154.
Compiler options

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

![Project>Options>C++ Compiler>Preprocessor>Preprocessor output to file](image)

---

**--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.

This option is not available in the IDE.

---

**-r, --debug**

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

**Description**
Use the `-r` or the `--debug` 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++ Compiler>Output>Generate debug information](image)

---

**--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 151.

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.

Note: This option only applies to functions in the C standard library.

Project>Options>C/C++ Compiler>Language>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 177, and Descriptions of predefined preprocessor symbols, page 284 for information about the preprocessor symbol __ROPI__.
--rwpi

Syntax
--rwpi

Description
Use this option to make the compiler generate code that uses the offset from the static base register (R9) to address-writable data.

When this option is used, these limitations apply:

- The object attribute __ramfunc cannot be used
- Pointer constants cannot be initialized with the address of a writable variable. However, static writable variables can be initialized to writable variable addresses at runtime.

See also
--no_rwlock_dynamic_init, page 177, and Descriptions of predefined preprocessor symbols, page 284 for information about the preprocessor symbol __RWPI__

--section

Syntax
--section OldName=NewName

Description
The compiler places functions and data objects into named sections which are referred to by the IAR ILINK Linker. Use this option to change the name of the section OldName to NewName.

This is useful if you want to place your code or data in different address ranges and you find the @ notation, alternatively the #pragma location directive, insufficient. Note that any changes to the section names require corresponding modifications in the linker configuration file.

Example
To place functions in the section MyText, use:
--section .text=MyText

See also
For information about the different methods for controlling placement of data and code, see Controlling data and function placement in memory, page 129.
--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.

--strict_ansi

Syntax  
--strict_ansi

Description
By default, the compiler accepts a relaxed superset of ISO/ANSI C/C++, see Minor language extensions, page 228. Use this option to ensure that the program conforms to the ISO/ANSI C/C++ standard.

Note: The -e option and the --strict_ansi option cannot be used at the same time.

Project>Options>C/C++ Compiler>Language>Language conformances>Strict ISO/ANSI
--thumb

Syntax
--thumb

Description
Use this option to set default function mode to Thumb. This setting must be the same for all files included in a program, unless they are interworking.

Note: This option has the same effect as the --cpu_mode=thumb option.

See also
--interwork, page 171 and __interwork, page 238.

--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 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.

--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.

See also

--diag_warning, page 225.

Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors
Linker options

This chapter gives detailed reference information about each linker option.

For general syntax rules, see Options syntax, page 153.

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>--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>--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>--define_symbol</td>
<td>Defines symbols that can be used by the application</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>--entry</td>
<td>Treats the symbol as a root symbol and as the start of the application</td>
</tr>
<tr>
<td>--error_limit</td>
<td>Specifies the allowed number of errors before compilation stops</td>
</tr>
<tr>
<td>--export_builtin_config</td>
<td>Produces an icf file for the default configuration</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line</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>--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>
</tbody>
</table>

Table 24: Linker options summary
<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>--no_fragments</td>
<td>Disables section fragment handling</td>
</tr>
<tr>
<td>--no_library_search</td>
<td>Disables automatic runtime library search</td>
</tr>
<tr>
<td>--no_locals</td>
<td>Removes local symbols from the ELF executable image.</td>
</tr>
<tr>
<td>--no_remove</td>
<td>Disables removal of unused sections</td>
</tr>
<tr>
<td>--no_veneers</td>
<td>Disables generation of veneers</td>
</tr>
<tr>
<td>--no_warnings</td>
<td>Disables generation of warnings</td>
</tr>
<tr>
<td>--no_wrap_diagnostics</td>
<td>Does not wrap long lines in diagnostic messages</td>
</tr>
<tr>
<td>-o</td>
<td>Sets the object filename</td>
</tr>
<tr>
<td>--only_stdout</td>
<td>Uses standard output only</td>
</tr>
<tr>
<td>--ose_load_module</td>
<td>Produces an OSE load module image</td>
</tr>
<tr>
<td>--output</td>
<td>Sets the object filename</td>
</tr>
<tr>
<td>--pi_veneers</td>
<td>Generates position independent veneers.</td>
</tr>
<tr>
<td>--place_holder</td>
<td>Reserve a place in ROM to be filled by some other tool, for example a checksum calculated by ielftool.</td>
</tr>
<tr>
<td>--redirect</td>
<td>Redirects a reference to a symbol to another symbol</td>
</tr>
<tr>
<td>--remarks</td>
<td>Enables remarks</td>
</tr>
<tr>
<td>--semihosting</td>
<td>Links with debug interface</td>
</tr>
<tr>
<td>--silent</td>
<td>Sets silent operation</td>
</tr>
</tbody>
</table>

Table 24: Linker options summary (Continued)
Descriptions of 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.

**--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**


**Project>Options>General Options>Target>Endian mode**

**--BE32**

**Syntax**

```
--BE32
```

**Description**

Use this option to specify the legacy big-endian mode.

---

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--strip</td>
<td>Removes debug information from the executable image</td>
</tr>
<tr>
<td>--warnings_are_errors</td>
<td>Warnings are treated as errors</td>
</tr>
<tr>
<td>--warnings_affect_exit_code</td>
<td>Warnings affects exit code</td>
</tr>
</tbody>
</table>

*Table 24: Linker options summary (Continued)*
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


---

**--config**

**Syntax**

```
--config filename
```

**Parameters**

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 154.

**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*.

---

**--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 that once on the command line.

See also

--define_symbol, page 193 and *Interaction between ILINK and the application*, page 55.
--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

core Specifies a specific processor variant

Description

Use this option to select the processor variant for which the code is to be generated. The default is ARM7TDMI.

See also

--cpu, page 161 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. By default, the value 0 (zero) is used.

constant_value The constant value of the symbol.
Descriptions of options

**Description**
Use this option to define a constant symbol that can be used by your application. If no value is specified, 0 is used. This option can be used more than once on the command line. Note that his option is different from the `define` symbol directive.

**See also**
`--config_def`, page 192 and `Interaction between ILINK and the application`, page 55.

**Project>Options>Linker>#define>Defined symbols**

---

**--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 violation of the C or C++ language rules, of such severity that a violation of the linking rules 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

\textit{tag} \hspace{1cm} 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.

\hspace{1cm} \rightarrow \textit{Project>Options>Linker>Diagnostics>Suppress these diagnostics}

--diag_warning

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

Parameters

\textit{tag} \hspace{1cm} 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.

\hspace{1cm} \rightarrow \textit{Project>Options>Linker>Diagnostics>Treat these as warnings}

--diagnostics_tables

Syntax
--diagnostics_tables \{filename|directory\}

Parameters
For information about specifying a filename or a directory, see \textit{Rules for specifying a filename or directory as parameters}, page 154.

Description
Use this option to list all possible diagnostic messages in a named file. This option cannot be given together with other options.

\hspace{1cm} \rightarrow \textit{This option is not available in the IDE.}
Descriptions of options

--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.

--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 154.

Description:
Exports the configuration used by default to a file.

This option is not available in the IDE.
Linker options

-\textit{-f}

\textbf{Syntax} \hspace{1cm} \textit{-f} \textit{filename}

\textbf{Parameters} \hspace{1cm} For information about specifying a filename, see \textit{Rules for specifying a filename or directory as parameters}, page 154.

\textbf{Descriptions} \hspace{1cm} Use this option to make the linker read command line options from the named file, with the default filename extension \textit{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 \texttt{Project}>\texttt{Options}>\texttt{Linker}>\texttt{Extra Options}.

\textbf{--force_output}

\textbf{Syntax} \hspace{1cm} \texttt{--force_output}

\textbf{Description} \hspace{1cm} Use this option to produce an output executable image regardless of any linking errors.

To set this option, use \texttt{Project}>\texttt{Options}>\texttt{Linker}>\texttt{Extra Options}

\textbf{--image_input}

\textbf{Syntax} \hspace{1cm} \texttt{--image_input} \textit{filename} \texttt{[symbol,]} \texttt{[section[,alignment]]]

\textbf{Parameters} \hspace{1cm}

\begin{itemize}
  \item \textit{filename} \hspace{2cm} The pure binary file containing the raw image you want to link
  \item \textit{symbol} \hspace{2cm} The symbol which the binary data can be referenced with.
  \item \textit{section} \hspace{2cm} The section where the binary data will be placed; default is \texttt{.text}.
  \item \textit{alignment} \hspace{2cm} The alignment of the section; default is \texttt{1}.
\end{itemize}

\textbf{Description} \hspace{1cm} Use this option to link pure binary files in addition to the ordinary input files. The file's entire contents are placed in the section, which means it can only contain pure binary data.
The section where the contents of the `filename` file are placed, is only included if the symbol `symbol` is required by your application. Use the `--keep` option if you want to force a reference to the section.

**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 198.

---

### `--keep`

**Syntax**

```
--keep symbol
```

**Parameters**

- `symbol`
  
The name of the symbol to be treated as a root symbol.

**Description**

Normally, the linker keeps a symbol only if it is needed by your application. Use this option to make a symbol always be included in the final application.

---

### `--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.
Part 2. Reference information

Linker options

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. You can specify more than one topic in a single use of the --log option, alternatively use separate --log options for different topics.

See also
--log_file, page 199.

--log_file
Syntax
--log_file filename
Parameters
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 154.
Description
Use this option to direct the log output to the specified file.
See also
--log, page 198.

--mangled_names_in_messages
Syntax
--mangled_names_in_messages
Descriptions
Use this option to produce both mangled and unmangled names for C/C++ symbols in messages. Mangling is a technique used for mapping a complex C name or a C++ name (for example, for overloading) into a simple name. For example, void h(int, char) becomes _Z1hicz.

This option is not available in the IDE.
Descriptions of options

**--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 AEABI attributes and IAR-specific 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

**--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.
---

**Linker options**

See also *Keeping symbols and sections*, page 51.

To set this option, use **Project>Options>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.

---

**--no_locals**

- **Syntax**: `--no_locals`
- **Description**: Use this option to remove local symbols from the ELF executable image.

**Note**: This option does not remove any local symbols from the DWARF information in the executable image.

---

**--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 51.

To set this option, use **Project>Options>Linker>Extra Options**
Descriptions of 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 56.

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

--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.

--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, --output
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 154.
Linker options

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

--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.

--ose_load_module
Syntax
--ose_load_module
Description
By default, the linker generates a ROMable executable image. Use this option to generate an executable image in the OSE load module image format instead.

Project>Options>Linker>Output

--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 154.
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
Descriptions of options

--pi_veneers

Syntax

```
--pi_veneers
```

Description
Use this option to make the linker generate position-independent veneers. Note that this type of veneers is bigger and slower than normal veneers.

See also
`Veneers`, page 56.

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

--place_holder

Syntax

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>symbol</code></td>
<td>The name of the symbol to create</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Size in ROM; by default 4 bytes</td>
</tr>
<tr>
<td><code>section</code></td>
<td>Section name to use; by default .text</td>
</tr>
<tr>
<td><code>alignment</code></td>
<td>Alignment of section; by default 1</td>
</tr>
</tbody>
</table>

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 325.

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 151.

--semihosting
Syntax
--semihosting[=iar_breakpoint]
Parameters
iar_breakpoint The IAR-specific mechanism can be used when debugging applications that use SWI/SVC extensively.
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
Low-level interface for debug support, page 62.

Project>Options>General Options>Library Configuration>Semihosted
Descriptions of options

--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.

--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

--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` directive 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 217.

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 only 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.

### 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 168.

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 212.

**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 ISO/ANSI 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>
<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>
</tbody>
</table>

*Table 25: 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};
```

Read also about the compiler option `--enum_is_int`, page 169.

**The char type**

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

**The wchar_t type**

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

---

### Table 25: Integer types (Continued)

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size</th>
<th>Range</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<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>0 to 2^{32} - 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>

Signed variables are represented using the two’s complement form.
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 ISO/ANSI C, int, signed int, and unsigned int can be used as the base type for integer bitfields. It is implementation defined whether the type specified by int is the same as signed int or unsigned int. In the IAR C/C++ Compiler for ARM, bitfields specified as int are treated as unsigned int. Furthermore, any integer type can be used as the base type when language extensions are enabled. Bitfields in expressions will have the same data type as the integer base type.

The compiler places bitfield members based on the byte order mode that is used. By default in little-endian mode, the compiler places bitfield members from the least significant to the most significant bit in the container type. And by default in big-endian mode, the compiler places bitfield members from the most significant to the least significant bit in the container type. A bitfield is assigned to the last available container of its base type which has enough unassigned bits to contain the entire bitfield. This means that bitfield containers can overlap other structure members as long as the order of the fields in the structure is preserved, for example in big-endian mode:

```
struct example
{
    char  a;
    short b : 10;
    int   c : 6;
};
```

Here the first declaration creates an unsigned character which is allocated to bits 24 through 31. The second declaration creates a signed short integer member of size 10 bits. This member is allocated to bits 15 through 6 as it will not fit in the remaining 8 bits of the first short integer container. The last bitfield member declared is placed in the bits 0 through 5. If seen as a 32-bit value, the structure looks like this in memory:

```
    7  0  7  0  15  6  5  0
   |   |   | 1  2  3
  a  pad b   c
```
Use the directive `#pragma bitfields=disjoint_types` to force the bitfield containers to be disjoint, or in other words, not to overlap. The layout of the above example structure would then become:

```
  7  0  7  0  15  6  5  0  31  26  25  0
a | pad b | pad c | pad
```

*Figure 15: Layout of bitfield members forced to be disjoint in big-endian mode*

Use the directive `#pragma bitfields=reversed_disjoint_types` to place the bitfield members from the least significant bit to the most significant bit in non-overlapping storage containers.

**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>Mantissa</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>
</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>
</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>
</tr>
</tbody>
</table>

*Table 26: 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 214.

Exception flags according to the IEEE 754 standard are not supported. The alignment for the `float` type is 4, and for the `long double` type it is 8.

**32-bit floating-point format**

The representation of a 32-bit floating-point number as an integer is:

```
 31 30 23 22 0
 S | Exponent | Mantissa
```

The exponent is 8 bits, and the mantissa is 23 bits.
The value of the number is:
\((-1)^S \times 2^{(Exponent -127)} \times 1.Mantissa\)

The range of the number is:
\([-1.18E-38 \text{ to } 3.40E+38]\)

The precision of the float operators (+, -, *, and /) is approximately 7 decimal digits.

### 64-bit floating-point format

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

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

The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:
\((-1)^S \times 2^{(Exponent -1023)} \times 1.Mantissa\)

The range of the number is:
\([-2.23E-308 \text{ to } 1.79E+308]\)

The precision of the float operators (+, -, *, and /) is approximately 15 decimal digits.

### Representation of special floating-point numbers

This list describes the representation of special floating-point numbers:

- **Zero** is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- **Infinity** is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- **Not a number** (NaN) is represented by setting the exponent to the highest positive value and at least one bit set in the 20 most significant bits of the mantissa. Remaining bits are zero.
- **Subnormal numbers** are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is denormalized, even though the number is treated as if the exponent was 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a denormalized number is:
  
  \((-1)^S \times 2^{(1-BIAS)} \times 0.Mantissa\)
where $\text{BIAS}$ is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

**Pointer types**

The compiler has two basic types of pointers: function pointers and data pointers.

**FUNCTION POINTERS**

The size of function pointers is always 32 bits and the range is $0x0-0xFFFFFFFF$.

When function pointer types are declared, attributes are inserted before the `*` sign, for example:

```c
typedef void (__thumb __interwork * IntHandler) (void);
```

This can be rewritten using `#pragma` directives:

```c
#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.
Structure types

**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 16: Structure layout](image)
The alignment of the structure is 2 bytes, and a pad byte must be inserted to give shorts the correct alignment.

**PACKED STRUCTURE TYPES**

The `__packed` data type attribute or the `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work.

**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 17: 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;
};
```
Type qualifiers

S2 has this memory layout

Figure 18: Packed structure layout

The structure S will use the memory layout, size, and alignment described in the previous example. The alignment of the member l is 4, which means that alignment of the structure S2 will become 2.

For more information, see Alignment of elements in a structure, page 127.

Type qualifiers

According to the ISO/ANSI C standard, volatile and const are type qualifiers.

DECLARING OBJECTS VOLATILE

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 ISO/ANSI 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.
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 other object types, only the rule that states that all accesses are preserved applies.

**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.
Data types in C++
Compiler extensions

This chapter gives a brief overview of the compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI 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.

You can find the extensions available as:

- C/C++ language extensions

  For a summary of available language extensions, see C language extensions, page 222. For reference 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 the ISO/ANSI C standard 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 a list of available pragma directives, see the chapterPragma directives.

- Preprocessor extensions

  The preprocessor of the compiler adheres to the ISO/ANSI standard. 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
C language extensions

instructions. To read more about using intrinsic functions, see Mixing C and assembler, page 89. For a list of available functions, see the chapter Intrinsic functions.

Library functions

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. The library also provides some extensions, partly taken from the C99 standard. For more information, see IAR DLIB Library, page 292.

Note: Any use of these extensions, except for the pragma directives, makes your application inconsistent with the ISO/ANSI C standard.

ENABLING LANGUAGE EXTENSIONS

In the IDE, language extensions are enabled by default.

For information about how to enable and disable language extensions from the command line, see the compiler options -e, page 167, and --strict_ansi, page 186.

C language extensions

This section gives a brief overview of the C language extensions available in the compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions are grouped according to their expected usefulness. In short, this means:

- Important language extensions—extensions specifically tailored for efficient embedded programming, typically to meet memory restrictions
- Useful language extensions—features considered useful and typically taken from related standards, such as C99 and C++
- Minor language extensions, that is, the relaxation of some minor standards issues and also some useful but minor syntax extensions.

IMPORTANT LANGUAGE EXTENSIONS

The following language extensions available both in the C and the C++ programming languages 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
DARM-5

Part 2. Reference information

Compiler extensions

section. For more information about using these primitives, see *Controlling data and function placement in memory*, page 129, and *location*, page 251.

- **Alignment**

  Each data type has its own alignment, for more details, see *Alignment*, page 209. 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 use 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 named 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 127.

- **Bitfields and non-standard types**

  In ISO/ANSI C, a bitfield must be of type int or unsigned int. Using IAR Systems language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 212.

**Dedicated section operators**

The compiler supports for these built-in section operators: **_section_begin**, **_section_end**, and **_section_size**.

These operators behave syntactically as if declared like this:

```c
void * _section_begin(char const * section)
void * _section_end(char const * section)
size_t * _section_size(char const * section)
```

These operators can be used on named sections or on named blocks defined in the linker configuration file.

The **_section_begin** operator returns the address of the first byte of the named section or block.

The **_section_end** operator returns the address of the first byte after the named section or block.

The **_section_size** operator returns the size of the named section or block in bytes.
Note: The aliases __segment_begin/__sfb, __segment_end/__sfe, and __segment_size/__sfs can also be used.

When using the @ operator or the #pragma location directive to place a data object or a function in a user-defined section, or when using named blocks in the linker configuration file, the section operators can be used for getting the start and end of the memory range where the sections or blocks were placed.

The named section must be a string literal and section 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>

Table 27: Section operators and their symbols

Note that the linker will not necessarily place sections with the same name contiguously 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 contiguously, 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"
...
section_start_address = __section_begin("MYSECTION");
```

See also section, page 256, and location, page 251.
USEFUL LANGUAGE EXTENSIONS

This section lists and briefly describes useful extensions, that is, useful features typically taken from related standards, such as C99 and C++:

● Inline functions

The `#pragma inline` directive, alternatively the `inline` keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword `inline`. For more information, see `inline`, page 250.

● Mixing declarations and statements

It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.

● Declaration in `for` loops

It is possible to have a declaration in the initialization expression of a `for` loop, for example:

```c
for (int i = 0; i < 10; ++i)
{  
    /* Do something here. */
}
```

This feature is part of the C99 standard and C++.

● The `bool` data type

To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)

● C++ style comments

C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

```c
// The length of the bar, in centimeters.  
int length;
```

This feature is copied from the C99 standard and C++.

**Inline assembler**

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict_ansi` is used. The `__asm` keyword is always available.

**Note:** Not all assembler directives or operators can be inserted using this keyword.
C language extensions

The syntax is:

```c
asm (*string*);
```

The string can be a valid assembler instruction or a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```c
asm (*Label: nop\n
   *  
   b Label*);
```

where \n (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions.

For more information about inline assembler, see Mixing C and assembler, page 89.

**Compound literals**

To create compound literals you can use this syntax:

```c
/* Create a pointer to an anonymous array */
int *p = (int []) {1, 2, 3};

/* Create a pointer to an anonymous structX */
structX *px = &(structX) {5, 6, 7};
```

**Note:**
- A compound literal can be modified unless it is declared `const`
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

**Incomplete arrays at end of structs**

The last element of a `struct` can be an incomplete array. This is useful for allocating a chunk of memory that contains both the structure and a fixed number of elements of the array. The number of elements can vary between allocations.

This feature is part of the C99 standard.

**Note:** The array cannot be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

**Example**

```c
struct str
{
    char a;
    unsigned long b[];
};
```
struct str * GetAStr(int size)
{
    return malloc(sizeof(struct str) +
                   sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
    s->b[10] = 0;
}

The incomplete array will be aligned in the structure just like any other member of the structure. For more information about structure alignment, see Structure types, page 216.

**Hexadecimal floating-point constants**

Floating-point constants can be given in hexadecimal style. The syntax is $0xMANTp(+|-)EXP$ where $MANT$ is the mantissa in hexadecimal digits, including an optional . (decimal point), and $EXP$ is the exponent with decimal digits, representing an exponent of 2. This feature is part of the C99 standard.

**Examples**

- $0x1p0$ is 1
- $0xA.8p2$ is $10.5 \times 2^2$

**Designated initializers in structures and arrays**

Any initialization of either a structure (struct or union) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as `.elementname` and for an array [constant index expression]. Using designated initializers is not supported in C++.
Examples

This definition shows a `struct` and its initialization using designators:

```c
struct {
    int i;
    int j;
    int k;
    int l;
    short array[10];
} u =
{
    .l = 6,          /* initialize l to 6 */
    .j = 6,          /* initialize j to 6 */
    8,               /* initialize k to 8 */
    .array[7] = 2,   /* initialize element 7 to 2 */
    .array[3] = 2,   /* initialize element 3 to 2 */
    5,               /* array[4] = 5 */
    .k = 4           /* reinitialize k to 4 */
};
```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```c
union {
    int i;
    float f;
} y = {.f = 5.0};
```

To set the size of an array by initializing the last element, do like this:

```c
char array[] = {{10} = 'a'};
```

MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards 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 IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- **Missing semicolon at end of `struct` or `union` specifier**
  A warning 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 ISO/ANSI C, some operators allow such things, while others do not allow them.

- **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 215.

- **Taking the address of a register variable**
  In ISO/ANSI 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.

- **Duplicated size and sign specifiers**
  Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error 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 strict ISO/ANSI 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 ISO/ANSI mode, a warning is issued.

● A label preceding a }
In ISO/ANSI 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 issues a warning.

**Note:** 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
ISO/ANSI 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, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
void func(char)
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 167.
Extended keywords

This chapter describes the extended keywords that support specific features of the ARM core and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The compiler provides a set of attributes that can be used on functions or data objects to support specific features of the ARM core. There are two types of attributes—type attributes and object attributes:

- Type attributes affect the external functionality of the data object or function
- Object attributes affect the internal functionality of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For detailed information about each attribute, see Descriptions of extended keywords, page 237.

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 167 for additional 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 directly in your source code, or use the pragma directive #pragma type_attribute.

These general type attributes are available:

- Function type attributes affect how the function should be called: __arm, __fiq, __interwork, __irq, __swi, __task, and __thumb
● **Data type attributes**: `__big_endian`, `const`, `__little_endian`, `__packed`, and `volatile`

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

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 218.

**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. 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 __little_endian * p;          // The int object will be accessed in little endian byte order.
int * __little_endian p;          // The pointer will be accessed in little endian byte order.
__little_endian int * p;          // The pointer will be accessed in little endian byte order.
```

**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
void (__irq __arm my_handler)(void);

This declaration of my_handler is equivalent with the previous one:

#pragma type_attribute=__irq __arm
void my_handler(void);

**OBJECT ATTRIBUTES**

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables: __absolute and __no_init
- Object attributes that can be used for functions and variables: location, @, __root, and __weak.
- Object attributes that can be used for functions: __intrinsic, __nested, __noreturn, and __ramfunc.

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

For more information about location and @, see *Controlling data and function placement in memory*, page 129.

**Syntax for object attributes**

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

__no_init int myarray[10];

The #pragma object_attribute directive can also be used. This declaration is equivalent to the previous one:

#pragma object_attribute=__no_init
int myarray[10];

**Note:** Object attributes cannot be used in combination with the typedef keyword.
Summary of extended keywords

This table summarizes the extended keywords:

<table>
<thead>
<tr>
<th>Extended keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__absolute</td>
<td>Makes references to the object use absolute addressing</td>
</tr>
<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>Supports 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>__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 28: Extended keywords summary
**Descriptions of extended keywords**

These sections give detailed information about each extended keyword.

__absolute__

**Syntax**

Follows the generic syntax rules for object attributes that can be used on data, see *Object attributes*, page 235.

**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 233.

**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 238.

__big_endian__

**Syntax**

Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 233.

**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.
Descriptions of extended keywords

**Example**  
__big_endian long my_variable;

**See also**  
__little_endian, page 239.

**__fiq**

**Syntax**  
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

**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.

**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 233.

**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 233.

**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.
__little_endian

Syntax | Follows the generic syntax rules for type attributes that can be used on data objects, see Type attributes, page 233.

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.

Example | __little_endian long my_variable;

See also | __big_endian, page 237.

__nested

Syntax | Follows the generic syntax rules for object attributes that can be used on functions, see Object attributes, page 235.

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 33.

__no_init

Syntax | Follows the generic syntax rules for object attributes, see Object attributes, page 235.

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.

Example | __no_init int myarray[10];
See also

__noreturn

Syntax

Follows the generic syntax rules for object attributes, see Object attributes, page 235.

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

__noreturn void terminate(void);

__packed

Syntax

Follows the generic syntax rules for type attributes that can be used on data, see Type attributes, page 233.

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.

Example

extern void G(int, char);

__packed struct X {char ch; int i;}; /* No pad bytes */
void FOO(struct X * xp) /* No need for __packed here */ {
    int __packed * p2 = &xp->i; /* OK */
    char * p3 = &xp->ch; /* OK, char not affected */
    G(*p2, *p3);
}
See also  

pack, page 253.

__ramfunc

Syntax  

Follows the generic syntax rules for object attributes, see Object attributes, page 235.

Description  

The __ramfunc keyword makes a function execute in RAM. Two code sections will be created: one for the RAM execution, and one for the ROM initialization.

If a function declared __ramfunc tries to access ROM, the compiler will issue a warning. This behavior is intended to simplify the creation of upgrade routines, for instance, rewriting parts of flash memory. If this is not why you have declared the function __ramfunc, you can safely ignore or disable these warnings.

Functions declared __ramfunc are by default stored in the section named .textrw.

Example  

__ramfunc int FlashPage(char * data, char * page);

See also  

To read more about __ramfunc declared functions in relation to breakpoints, see the IAR Embedded Workbench® IDE User Guide for ARM®.

__root

Syntax  

Follows the generic syntax rules for object attributes, see Object attributes, page 235.

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  

To read more about root symbols and how they are kept, see Keeping symbols and sections, page 51.
**__swi**

**Syntax**

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

**Description**

The **__swi** keyword 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 SWC 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:

```c
#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 swi0x23_function */
```
Somewhere in your application source code, you define your software interrupt function:

```c
...__swi __arm int the_actual_swih0x23_function(int a, int b)
{
  ...
  return 42;
}
```

See also *Software interrupts*, page 34 and *Calling convention*, page 95.

__**task**

**Syntax**  
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

**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**

```c
__task void my_handler(void);
```

__**thumb**

**Syntax**  
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

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

Example

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

See also

```
__interwork, page 238.
```

**__weak**

Syntax

Follows the generic syntax rules for object attributes, see Object attributes, page 235.

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 the ISO/ANSI C standard 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>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>include_alias</td>
<td>Specifies an alias for an include file</td>
</tr>
<tr>
<td>inline</td>
<td>Inlines a function</td>
</tr>
<tr>
<td>language</td>
<td>Controls the IAR Systems language extensions</td>
</tr>
<tr>
<td>location</td>
<td>Specifies the absolute address of a variable, 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>
</tbody>
</table>

Table 29: Pragma directives summary
Note: For portability reasons, the pragma directives `alignment`, `baseaddr`, `codeseg`, `constseg`, `datasetg`, `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 (6.8.6), page 359.

---

**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.
Pragma directives

**Pragma directives**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>joined_types</strong></td>
<td>Bitfield members are placed depending on the byte order. Storage containers of bitfields will overlap other structure members. For more details, see Bitfields, page 212.</td>
</tr>
<tr>
<td><strong>reversed_disjoint_types</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>reversed</strong></td>
<td>This is an alias for reversed_disjoint_types.</td>
</tr>
<tr>
<td><strong>default</strong></td>
<td>Restores to default layout of bitfield members. The default behavior for the compiler is joined_types.</td>
</tr>
</tbody>
</table>

**Description**

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

**Example**

```c
#pragma bitfields=disjoint_types
/* Structure that uses disjoint types. */
{
  unsigned char error :1;
  unsigned char size  :4;
  unsigned short code :10;
}
#endif /* Section */
```

**See also**

[Bitfields, page 212.](#)

**data_align**

**Syntax**

```c
#pragma data_alignment=expression
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A constant which must be a power of two (1, 2, 4, etc.).</td>
</tr>
</tbody>
</table>

**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 `Pe117`.

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`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also

*Diagnostics*, page 150.

**diag_error**

Syntax

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

Parameters

- `tag` The number of a diagnostic message, for example the message number `Pe117`.

Description

Use this pragma directive to change the severity level to *error* for the specified diagnostics.

See also

*Diagnostics*, page 150.

**diag_remark**

Syntax

```
#pragma diag_remark=tag[,tag,...]
```

Parameters

- `tag` The number of a diagnostic message, for example the message number `Pe117`.
### Pragma directives

<table>
<thead>
<tr>
<th>Pragma directive</th>
<th>Description</th>
<th>Syntax</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>diag_suppress</strong></td>
<td>Use this pragma directive to change the severity level to reminder for the specified diagnostic messages.</td>
<td><code>#pragma diag_suppress=tag[,tag,...]</code></td>
<td><code>tag</code> (The number of a diagnostic message, for example the message number Pe117).</td>
</tr>
<tr>
<td><strong>diag_warning</strong></td>
<td>Use this pragma directive to change the severity level to warning for the specified diagnostic messages.</td>
<td><code>#pragma diag_warning=tag[,tag,...]</code></td>
<td><code>tag</code> (The number of a diagnostic message, for example the message number Pe826).</td>
</tr>
<tr>
<td><strong>include_alias</strong></td>
<td>Use this pragma directive to create an alias for the original header file.</td>
<td><code>#pragma include_alias (&quot;orig_header&quot;, &quot;subst_header&quot;)</code></td>
<td><code>orig_header</code> (The name of a header file for which you want to create an alias). <code>subst_header</code> (The alias for the original header file).</td>
</tr>
</tbody>
</table>

---

See also: *Diagnostics*, page 150.
**include_alias**

**Description**
Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

**Example**
```
#pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

**See also**
Include file search procedure, page 147.

**inline**

**Syntax**
```
#pragma inline [=forced]
```

**Parameters**
- `forced` Disables the compiler's heuristics and forces inlining.

**Description**
Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually occurs is subject to the compiler’s heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler’s heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

**Note:** Because specifying `#pragma inline=forced` disables the compiler’s heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.
Pragma directives

**language**

Syntax

```
#pragma language={extended|default}
```

Parameters

- `extended` Turns on the IAR Systems language extensions and turns off the `--strict_ansi` command line option.
- `default` Uses the language settings specified by compiler options.

Description

Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.

**location**

Syntax

```
#pragma location={address|NAME}
```

Parameters

- `address` The absolute address of the global or static variable for which you want an absolute location.
- `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 variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a section for placing either a variable or a function whose declaration follows the pragma directive.

Example

```
#pragma location=0xFFFF0400
__no_init volatile char PORT1; /* PORT1 is located at address 0xFFFF0400 */
```

```
#pragma location="foo"
char PORT1; /* PORT1 is located in section foo */
```

```
/* 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 129.
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:

```c
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

#pragma object_attribute=object_attribute[ ,object_attribute,...]

Parameters

For a list of object attributes that can be used with this pragma directive, see Object attributes, page 235.

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

```c
#pragma object_attribute=__no_init
char bar;
```

See also

General syntax rules for extended keywords, page 233.

optimize

Syntax

#pragma optimize=param[ param...]

Parameters

balanced|size|speed

Optimizes balanced between speed and size, optimizes for size, or optimizes for speed

none|low|medium|high

Specifies the level of optimization

no_code_motion

Turns off code motion
Pragma directives

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 speed, size, and balanced 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
#pragma optimize=speed
int small_and_used_often()
{
    ...
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
    ...
}

pack

Syntax
#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

Turns off common subexpression elimination
Turns off function inlining
Turns off type-based alias analysis
Turns off loop unrolling
Turns off instruction scheduling
Descriptions of pragma directives

#pragma pack

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 end of file.

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

See also Structure types, page 216 and `__packed`, page 240.

__printf_args

Syntax
#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.

Example
#pragma __printf_args
int printf(char const *,...);
/* Function call */
printf("%d",x);  /* Compiler checks that x is an integer */

required

Syntax
#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.
Pragma directives

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**

```
#pragma rtmodel="key","value"
```

**Parameters**

- **"key"**
  A text string that specifies the runtime model attribute.

- **"value"**
  A text string that specifies the value of the runtime model attribute. Using the special value "*" is equivalent to not defining the attribute at all.

**Description**

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value ". It can, however, be useful to state explicitly that the module can handle any runtime model. A module can have several runtime model definitions.

**Note:** The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

**Example**

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

**See also**

*Checking module consistency*, page 85.
__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;
}

section

Syntax

#pragma section="NAME" [align]

alias
#pragma segment="NAME" [align]

Parameters

NAME                          The name of the section or segment
align                        Specifies an alignment for the section. The value must be a constant integer expression to the power of two.

Description
Use this pragma directive to define a section name that can be used by the section operators __section_begin, __section_end, and __section_size. All section declarations for a specific section must have the same memory type attribute and alignment.

Example

#pragma section="MYSECTION" 4

See also
Dedicated section operators, page 223. For more information about sections, see the chapter Linking your application.
swi_number

Syntax
#pragma swi_number=number

Parameters
number The software interrupt number

Description
Use this pragma directive together with the __swi extended keyword. It is used as an argument to the generated SWC assembler instruction, and is used for selecting one software interrupt function in a system containing several such functions.

Example
#pragma swi_number=17

See also
Software interrupts, page 34.

type_attribute

Syntax
#pragma type_attribute=type_attribute[, type_attribute,...]

Parameters
For a list of type attributes that can be used with this pragma directive, see Type attributes, page 233.

Description
Use this pragma directive to specify IAR-specific type attributes, which are not part of the ISO/ANSI C language standard. 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
See the chapter Extended keywords for more details.
**weak**

**Syntax**

```
#pragma weak symbol1={symbol2}
```

**Parameters**

- `symbol1`: A function or variable with external linkage.
- `symbol2`: A defined function or variable.

**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:

```
#pragma weak foo
```

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

```
#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 244.
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`. Note that the intrinsic function names start with double underscores, for example:

```
__disable_interrupt
```

This table summarizes the intrinsic functions:

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>

Table 30: Intrinsic functions summary
<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__get_interrupt_state</td>
<td>Returns the interrupt state</td>
</tr>
<tr>
<td>__get_PRIMASK</td>
<td>Returns the value of the Cortex-M PRIMASK register</td>
</tr>
<tr>
<td>__ISB</td>
<td>Insert a ISB instruction</td>
</tr>
<tr>
<td>__LDC</td>
<td>Inserts the coprocessor load instruction LD C</td>
</tr>
<tr>
<td>__LDCCL</td>
<td>Inserts the coprocessor load instruction LDC</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_noidx</td>
<td>Inserts the coprocessor load instruction LDC</td>
</tr>
<tr>
<td>__LDCCL_noidx</td>
<td>Inserts the coprocessor load instruction LDC</td>
</tr>
<tr>
<td>__LDC2_noidx</td>
<td>Inserts the coprocessor load instruction LDC2</td>
</tr>
<tr>
<td>__LDC2L_noidx</td>
<td>Inserts the coprocessor load instruction LDC2L</td>
</tr>
<tr>
<td>__LDREX</td>
<td>Inserts an LDREX instruction</td>
</tr>
<tr>
<td>__MCR</td>
<td>Inserts the coprocessor write instruction MCR</td>
</tr>
<tr>
<td>__MRC</td>
<td>Inserts the coprocessor read instruction MRC</td>
</tr>
<tr>
<td>__no_operation</td>
<td>Inserts a NOP 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>__QDADD</td>
<td>Inserts a QDADD instruction</td>
</tr>
<tr>
<td>__QDOUBLE</td>
<td>Inserts a QDOUBLE instruction</td>
</tr>
<tr>
<td>__QSUB</td>
<td>Inserts a QSUB instruction</td>
</tr>
<tr>
<td>__QSUB8</td>
<td>Inserts a QSUB8 instruction</td>
</tr>
<tr>
<td>__QSUB16</td>
<td>Inserts a QSUB16 instruction</td>
</tr>
<tr>
<td>__QSAX</td>
<td>Inserts a QSAX instruction</td>
</tr>
<tr>
<td>__REV</td>
<td>Inserts a REV instruction</td>
</tr>
<tr>
<td>__REVSH</td>
<td>Inserts a REVSH instruction</td>
</tr>
<tr>
<td>__SADD8</td>
<td>Inserts a SADD8 instruction</td>
</tr>
<tr>
<td>__SADD16</td>
<td>Inserts a SADD16 instruction</td>
</tr>
</tbody>
</table>

Table 30: Intrinsic functions summary (Continued)
Intrinsic functions

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__SASX</td>
<td>Inserts a SASX instruction</td>
</tr>
<tr>
<td>__SEL</td>
<td>Inserts a SEL instruction</td>
</tr>
<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_interrupt_state</td>
<td>Restores the interrupt state</td>
</tr>
<tr>
<td>__set_PRIMASK</td>
<td>Sets the value of the Cortex-M PRIMASK register</td>
</tr>
<tr>
<td>__SHADD8</td>
<td>Inserts a SHADD8 instruction</td>
</tr>
<tr>
<td>__SHADD16</td>
<td>Inserts a SHADD16 instruction</td>
</tr>
<tr>
<td>__SHASX</td>
<td>Inserts a SHASX instruction</td>
</tr>
<tr>
<td>__SHSUB8</td>
<td>Inserts a SHSUB8 instruction</td>
</tr>
<tr>
<td>__SHSUB16</td>
<td>Inserts a SHSUB16 instruction</td>
</tr>
<tr>
<td>__SHSAX</td>
<td>Inserts a SHSAX instruction</td>
</tr>
<tr>
<td>__SMUL</td>
<td>Inserts a signed 16-bit multiplication</td>
</tr>
<tr>
<td>__SSUB8</td>
<td>Inserts a SSUB8 instruction</td>
</tr>
<tr>
<td>__SSUB16</td>
<td>Inserts a SSUB16 instruction</td>
</tr>
<tr>
<td>__SSAX</td>
<td>Inserts a SSAX 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>
<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>__SWP</td>
<td>Inserts an SWP instruction</td>
</tr>
<tr>
<td>__SWPB</td>
<td>Inserts an SWPB instruction</td>
</tr>
<tr>
<td>__UADD8</td>
<td>Inserts a UADD8 instruction</td>
</tr>
</tbody>
</table>

Table 30: Intrinsic functions summary (Continued)
Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

**__CLZ**

**Syntax**

```
unsigned char __CLZ(unsigned long);
```

**Description**

Inserts a CLZ instruction.

This intrinsic function requires an ARM v5 architecture or higher for ARM mode, and ARM v6T2 or higher for Thumb mode.

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>__UHSAAX</td>
<td>Inserts a UHSAAX 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>__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>__UQSUB8</td>
<td>Inserts a UQSUB8 instruction</td>
</tr>
<tr>
<td>__UQSUB16</td>
<td>Inserts a UQSUB16 instruction</td>
</tr>
<tr>
<td>__UQSAAX</td>
<td>Inserts a UQSAAX instruction</td>
</tr>
<tr>
<td>__UQSUB8</td>
<td>Inserts a UQSUB8 instruction</td>
</tr>
<tr>
<td>__USAX</td>
<td>Inserts a USAX instruction</td>
</tr>
<tr>
<td>__USUB8</td>
<td>Inserts a USUB8 instruction</td>
</tr>
<tr>
<td>__USUB16</td>
<td>Inserts a USUB16 instruction</td>
</tr>
</tbody>
</table>

Table 30: Intrinsic functions summary (Continued)
### __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 ARM v6M architecture, or an ARM v7 architecture or higher.

### __DSB

**Syntax**

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

**Description**

Inserts a DSB instruction. This intrinsic function requires an ARM v6M architecture, or an ARM v7 architecture or higher.
Descriptions of intrinsic functions

__enable_fiq
Syntax:
void __enable_fiq(void);

Description:
Enables fast interrupt requests (fiq).
This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

__enable_interrupt
Syntax:
void __enable_interrupt(void);

Description:
Enables interrupts. For Cortex-M devices, it resets the execution priority level to default by clearing the priority mask bit, PRIMASK. For other devices, it enables interrupt requests (irq) and fast interrupt requests (fiq).
This intrinsic function can only be used in privileged mode.

__enable_irq
Syntax:
void __enable_irq(void);

Description:
Enables interrupt requests (irq).
This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

__get_BASEPRI
Syntax:
unsigned long __get_BASEPRI(void);

Description:
Returns the value of the BASEPRI register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 or Cortex-M4 device.

__get_CONTROL
Syntax:
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**

```c
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.

### __get_FAULTMASK

**Syntax**

```c
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_interrupt_state

**Syntax**

```c
__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**

```c
__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_LR

**Syntax**

```c
unsigned long __get_LR(void);
```

**Description**

Returns the value of the link register (R14).
__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_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 a ISB instruction. This intrinsic function requires an ARM v6M architecture, or an ARM v7 architecture or higher.

__LDC  __LDCL  __LDC2  __LDC2L
Syntax
void __nnn(__ul coproc, __ul CRn, __ul const *src);
where nnn can be one of LDC, LDCL, LDC2, or LDC2L.
Parameters
coproc The coprocessor number 0..15.
CRn The coprocessor register to load.
src A pointer to the data to load.
Intrinsic functions

**Description**

Inserts the coprocessor load instruction \texttt{LDC}—or one of its variants—which means that a value will be loaded into a coprocessor register. The parameters \texttt{coproc} and \texttt{CRn} will be encoded in the instruction and must therefore be constants.

**\_\_LDC\_noidx**

**\_\_LDCL\_noidx**

**\_\_LDC2\_noidx**

**\_\_LDC2L\_noidx**

**Syntax**

```c
void \_\_nnn\_noidx(__ul coproc, __ul CRn, __ul const *src, __ul option);
```

where \texttt{nnn} can be one of \texttt{LDC}, \texttt{LDCL}, \texttt{LDC2}, or \texttt{LDC2L}.

**Parameters**

- \texttt{coproc} \hspace{1cm} The coprocessor number 0..15.
- \texttt{CRn} \hspace{1cm} The coprocessor register to load.
- \texttt{src} \hspace{1cm} A pointer to the data to load.
- \texttt{option} \hspace{1cm} Additional coprocessor option 0..255.

**Description**

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

**\_\_LDREX**

**Syntax**

```c
unsigned long \_\_LDREX(unsigned long *);
```

**Description**

Inserts an \texttt{LDREX} instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v6T2 or higher for Thumb mode.

**\_\_MCR**

**Syntax**

```c
void \_\_MCR(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul CRm, __ul opcode_2);
```

**Parameters**

- \texttt{coproc} \hspace{1cm} The coprocessor number 0..15.
Descriptions of intrinsic functions

__MRC

Syntax

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

Parameters

coprocn 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). Returns the value of the specified coprocessor register. The parameters coproc, opcode_1, CRn, CRm, and opcode_2 will be encoded in the MRC instruction operation code and must therefore be constants.

This intrinsic function requires either ARM mode, or an ARM v6T2 or higher for Thumb mode.

__no_operation

Syntax

void __no_operation(void);

Description

Inserts a NOP instruction.
**__QADD**

**Syntax**

```c
signed long __QADD(signed long, signed long);
```

**Description**

Inserts a \texttt{QADD} instruction.

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.

---

**__QADD8**

**Syntax**

```c
unsigned long __QADD8(unsigned long, unsigned long);
```

**Description**

Inserts a \texttt{QADD8} instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

---

**__QADD16**

**Syntax**

```c
unsigned long __QADD16(unsigned long, unsigned long);
```

**Description**

Inserts a \texttt{QADD16} instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

---

**__QASX**

**Syntax**

```c
unsigned long __QASX(unsigned long, unsigned long);
```

**Description**

Inserts a \texttt{QASX} instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

---

**__QDADD**

**Syntax**

```c
signed long __QDADD(signed long, signed long);
```

**Description**

Inserts a \texttt{QDADD} instruction.
Descriptions of intrinsic functions

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.

**__QDOUBLE**

Syntax

```c
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 an ARM v5E architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__QDSUB**

Syntax

```c
signed long __QDSUB(signed long, signed long);
```

Description

Inserts a `QDSUB` instruction.

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.

**__QFlag**

Syntax

```c
int __QFlag(void);
```

Description

Returns 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.

**__QSUB**

Syntax

```c
signed long __QSUB(signed long, signed long);
```

Description

Inserts a QSUB instruction.

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.
__QSUB8
Syntax
unsigned long __QSUB8(unsigned long, unsigned long);
Description
Inserts a QSUB8 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__QSUB16
Syntax
unsigned long __QSUB16(unsigned long, unsigned long);
Description
Inserts a QSUB16 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__QSAX
Syntax
unsigned long __QSAX(unsigned long, unsigned long);
Description
Inserts a QSAX instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__REV
Syntax
unsigned long __REV(unsigned long);
Description
Inserts a REV instruction. This intrinsic function requires an ARM v6 architecture or
higher.

__REVSH
Syntax
signed long __REVSH(short);
Description
Inserts a REVSH instruction. This intrinsic function requires an ARM v6 architecture or
higher.
Descriptions of intrinsic functions

**__SADD8**

**Syntax**

unsigned long __SADD8(unsigned long, unsigned long);

**Description**

Inserts a SADD8 instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__SADD16**

**Syntax**

unsigned long __SADD16(unsigned long, unsigned long);

**Description**

Inserts a SADD16 instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__SASX**

**Syntax**

unsigned long __SASX(unsigned long, unsigned long);

**Description**

Inserts a SASX instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__SEL**

**Syntax**

unsigned long __SEL(unsigned long, unsigned long);

**Description**

Inserts a SEL instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-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__interrupt_state

Syntax
void __set__interrupt_state(__istate_t);

Descriptions
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 265.

__set__LR

Syntax
void __set__LR(unsigned long);

Description
Assigns a new address to the link register (R14).
Descriptions of intrinsic functions

__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_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).

__SHADD8
Syntax
unsigned long __SHADD8(unsigned long, unsigned long);
Description
Inserts a SHADD8 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__SHADD16
Syntax
unsigned long __SHADD16(unsigned long, unsigned long);
Description
Inserts a SHADD16 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.
__SHASX
Syntax
unsigned long __SHASX(unsigned long, unsigned long);
Description
Inserts a SHASX instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__SHSUB8
Syntax
unsigned long __SHSUB8(unsigned long, unsigned long);
Description
Inserts a SHSUB8 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__SHSUB16
Syntax
unsigned long __SHSUB16(unsigned long, unsigned long);
Description
Inserts a SHSUB16 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__SHSAX
Syntax
unsigned long __SHSAX(unsigned long, unsigned long);
Description
Inserts a SHSAX instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__SMUL
Syntax
signed long __SMUL(signed short, signed short);
Description
Inserts a signed 16-bit multiplication.
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.

__SSUB8
Syntax
unsigned long __SSUB8(unsigned long, unsigned long);
Description
Inserts a SSUB8 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__SSUB16
Syntax
unsigned long __SSUB16(unsigned long, unsigned long);
Description
Inserts a SSUB16 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__SSAX
Syntax
unsigned long __SSAX(unsigned long, unsigned long);
Description
Inserts a SSAX instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__STC
__STCL
__STC2
__STC2L
Syntax
void __nnn(__ul coproc, __ul CRn, __ul const *dst);
where nnn can be one of STC, STCL, STC2, or STC2L.
Parameters
coproc The coprocessor number 0..15.
Intrinsic functions

---

### __STC_noidx__, __STCL_noidx__, __STC2_noidx__, __STC2L_noidx__

**Syntax**

```c
void __nnn_noidx(__ul coproc, __ul CRn, __ul const *dst, __ul option);
```

where `nnn` can be one of `STC`, `STCL`, `STC2`, or `STC2L`.

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

---

### __STREX__

**Syntax**

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

**Description**

Inserts a `STREX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v6T2 or higher for Thumb mode.

---

- `CRn` The coprocessor register to load.
- `dst` A pointer to the destination.
- `option` Additional coprocessor option 0..255.
### ___SWP___

**Syntax**

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

**Description**

Inserts an `SWP` instruction. This intrinsic function requires ARM mode.

### ___SWPB___

**Syntax**

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

**Description**

Inserts an `SWPB` instruction. This intrinsic function requires ARM mode.

### ___UADD8___

**Syntax**

`unsigned long __UADD8(unsigned long, unsigned long);`

**Description**

Inserts a `UADD8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

### ___UADD16___

**Syntax**

`unsigned long __UADD16(unsigned long, unsigned long);`

**Description**

Inserts a `UADD16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

### ___UASX___

**Syntax**

`unsigned long __UASX(unsigned long, unsigned long);`

**Description**

Inserts a `UASX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and
ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.
Intrinsic functions

__UHADD8
Syntax
unsigned long __UHADD8(unsigned long, unsigned long);
Description
Inserts a UHADD8 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__UHADD16
Syntax
unsigned long __UHADD16(unsigned long, unsigned long);
Description
Inserts a UHADD16 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__UHASX
Syntax
unsigned long __UHASX(unsigned long, unsigned long);
Description
Inserts a UHASX instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__UHSAX
Syntax
unsigned long __UHSAX(unsigned long, unsigned long);
Description
Inserts a UHSAX instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__UHSUB8
Syntax
unsigned long __UHSUB8(unsigned long, unsigned long);
Description
Inserts a UHSUB8 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__UHSUB16**

Syntax: `unsigned long __UHSUB16(unsigned long, unsigned long);`

Description: Inserts a UHSUB16 instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__UQADD8**

Syntax: `unsigned long __UQADD8(unsigned long, unsigned long);`

Description: Inserts a UQADD8 instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__UQADD16**

Syntax: `unsigned long __UQADD16(unsigned long, unsigned long);`

Description: Inserts a UQADD16 instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

**__UQASX**

Syntax: `unsigned long __UQASX(unsigned long, unsigned long);`

Description: Inserts a UQASX instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.
__UQSUB8
Syntax
unsigned long __UQSUB8(unsigned long, unsigned long);
Description
Inserts a UQSUB8 instruction. This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__UQSUB16
Syntax
unsigned long __UQSUB16(unsigned long, unsigned long);
Description
Inserts a UQSUB16 instruction. This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__UQSAX
Syntax
unsigned long __UQSAX(unsigned long, unsigned long);
Description
Inserts a UQSAX instruction. This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__USAX
Syntax
unsigned long __USAX(unsigned long, unsigned long);
Description
Inserts a USAX instruction. This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__USUB8
Syntax
unsigned long __USUB8(unsigned long, unsigned long);
Description
Inserts a USUB8 instruction.
This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M for Thumb mode.

__USUB16

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long __USUB16(unsigned long, unsigned long);</td>
<td>Inserts a USUB16 instruction.</td>
</tr>
</tbody>
</table>

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7A, ARM v7R, or ARM v7E-M 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 the ISO/ANSI standard. 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 details, see *Descriptions of predefined preprocessor symbols*, page 284.

- **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 162.

- **Preprocessor extensions**
  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 287.

- **Preprocessor output**
  Use the option `--preprocess` to direct preprocessor output to a named file, see `--preprocess`, page 182.

Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 358.
This table describes the predefined preprocessor symbols:

<table>
<thead>
<tr>
<th>Predefined symbol</th>
<th>Identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARM_PROFILE_M</strong></td>
<td>An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture is a profile M core. The symbol is undefined for other cores.</td>
</tr>
<tr>
<td><strong>ARMVFP</strong></td>
<td>An integer that reflects the <code>--fpu</code> option and is defined to <strong>ARMVFPV1</strong>, <strong>ARMVFPV2</strong>, <strong>ARMVFPV3</strong>, or <strong>ARMVFPV4</strong>. These symbolic names can be used when testing the <strong>ARMVFP</strong> symbol. If VFP code generation is disabled (default), the symbol will be undefined.</td>
</tr>
<tr>
<td><strong>ARMVFP_D16</strong></td>
<td>An integer that is set based on the <code>--fpu</code> option. The symbol is set to 1 if the selected FPU is a VFPv3 or VFPv4 unit with only 16 D-registers. Otherwise, the symbol is undefined.</td>
</tr>
<tr>
<td><strong>ARMVFP_FP16</strong></td>
<td>An integer that is set based on the <code>--fpu</code> option. The symbol is set to 1 if the selected FPU only supports 16-bit floating-point numbers. Otherwise, the symbol is undefined.</td>
</tr>
<tr>
<td><strong>ARMVFP_SP</strong></td>
<td>An integer that is set based on the <code>--fpu</code> option. The symbol is set to 1 if the selected FPU only supports single-precision. Otherwise, the symbol is undefined.</td>
</tr>
<tr>
<td><strong>BASE_FILE</strong></td>
<td>A string that identifies the name of the base source file (that is, not the header file), being compiled. See also <strong>FILE</strong>, page 285, and --no_path_in_file_macros, page 176.</td>
</tr>
<tr>
<td><strong>BUILD_NUMBER</strong></td>
<td>A unique integer that identifies the build number of the compiler currently in use.</td>
</tr>
<tr>
<td><strong>CORE</strong></td>
<td>An integer that identifies the processor architecture in use. The symbol reflects the <code>--cpu</code> option and is defined to <strong>ARM4M</strong>, <strong>ARM4TM</strong>, <strong>ARM5</strong>, <strong>ARM5E</strong>, <strong>ARM6</strong>, <strong>ARM6M</strong>, <strong>ARM6SM</strong>, <strong>ARM7M</strong>, <strong>ARM7EM</strong>, or <strong>ARM7R</strong>. These symbolic names can be used when testing the <strong>CORE</strong> symbol.</td>
</tr>
<tr>
<td>__cplusplus</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 31: Predefined symbols
<table>
<thead>
<tr>
<th>Predefined symbol</th>
<th>Identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU_MODE</strong></td>
<td>An integer that reflects the selected CPU mode and is defined to 1 for Thumb and 2 for ARM.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>A string that identifies the date of compilation, which is returned in the form &quot;Mmm dd yyyy&quot;, for example &quot;Oct 30 2008&quot;.</td>
</tr>
<tr>
<td>__embedded_cplusplus</td>
<td>An integer which is defined to 1 when the compiler runs in any of the C++ modes, 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.</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also <strong>BASE_FILE</strong>, page 284, and --no_path_in_file_macros, page 176.</td>
</tr>
<tr>
<td><strong>func</strong></td>
<td>A string that identifies 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 -e, page 167. See also <strong>PRETTY_FUNCTION</strong>, page 286.</td>
</tr>
<tr>
<td><strong>FUNCTION</strong></td>
<td>A string that identifies 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 -e, page 167. See also <strong>PRETTY_FUNCTION</strong>, page 286.</td>
</tr>
<tr>
<td><strong>IAR_SYSTEMS_ICC</strong></td>
<td>An integer that identifies the IAR compiler platform. The current value is 7. 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.</td>
</tr>
<tr>
<td><strong>ICCARM</strong></td>
<td>An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for ARM, and otherwise to 0.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>LITTLE_ENDIAN</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 31: Predefined symbols  (Continued)
Descriptions of predefined preprocessor symbols

<table>
<thead>
<tr>
<th>Predefined symbol</th>
<th>Identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PRETTY_FUNCTION</strong></td>
<td>A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example &quot;void func(char)&quot;. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 167. See also <strong>func</strong>, page 285.</td>
</tr>
<tr>
<td><strong>ROPI</strong></td>
<td>An integer that is defined when the --ropi compiler option is used; see --ropi, page 184.</td>
</tr>
<tr>
<td><strong>RWPI</strong></td>
<td>An integer that is defined when the --rwpi compiler option is used; see --rwpi, page 185.</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>An integer that is set to 1, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with #ifdef to detect whether the compiler in use adheres to ISO/ANSI C.</td>
</tr>
<tr>
<td><strong>STDC_VERSION</strong></td>
<td>An integer that identifies the version of ISO/ANSI C standard in use. The symbols expands to 199409L. This symbol does not apply in EC++ mode.</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>
</tr>
<tr>
<td><strong>VER</strong></td>
<td>An integer that identifies the version number of the IAR compiler in use. For example, version 5.11.3 is returned as 5011003.</td>
</tr>
</tbody>
</table>

Table 31: Predefined symbols (Continued)

* This symbol is required by the ISO/ANSI standard.

**TID**

Description

Target identifier for the IAR C/C++ Compiler for ARM. Expands to the target identifier which contains the following parts:

- A one-bit intrinsic flag (i) which is reserved for use by IAR
- A target identifier (t) unique for each IAR compiler. For the ARM compiler, the target identifier is 79
- A value (c) reserved for specifying different CPU core families. The value is derived from the setting of the --cpu option:

<table>
<thead>
<tr>
<th>Value</th>
<th>CPU core family</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unspecified</td>
</tr>
</tbody>
</table>

Table 32: Values for specifying different CPU core families in **TID**
The __TID__ value is constructed as:

\[
((i << 15) | (t << 8) | (c << 4))
\]

You can extract the values as follows:

\[
i = (__TID__ >> 15) & 0x01; /* intrinsic flag */
\]

\[
t = (__TID__ >> 8) & 0x7F; /* target identifier */
\]

\[
c = (__TID__ >> 4) & 0x0F; /* cpu core family */
\]

To find the value of the target identifier for the current compiler, execute:

```
printf("%ld",(__TID__ >> 8) & 0x7F)
```

Note: Because coding may change or functionality may be entirely removed in future versions, the use of __TID__ is not recommended. We recommend that you use the symbols __ICCARM__ and __CORE__ instead.

## 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 ISO/ANSI directives.

### NDEBUG

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 83.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

### _Pragma()

**Syntax**

```
Pragma("string")
```

where `string` follows the syntax of the corresponding pragma directive.

**Description**

This preprocessor operator is part of the C99 standard and can be used, for example, in defines and is equivalent to the `#pragma` directive.

**Note:** The `-e` option—enable language extensions—does not have to be specified.

**Example**

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=none")
#else
  #define NOOPT
#endif
```

See also

- See the chapter [Pragma directives](#).

### #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 ISO/ANSI standard `#error` directive is used.
__VA_ARGS__

Syntax

```c
#define P(...)       __VA_ARGS__
#define P(x, y, ...)   x + y + __VA_ARGS__
```

__VA_ARGS__ will contain all variadic arguments concatenated, including the separating commas.

Description

Variadic macros are the preprocessor macro equivalents of printf style functions. __VA_ARGS__ is part of the C99 standard.

Example

```c
#if DEBUG
#define DEBUG_TRACE(S, ...) printf(S, __VA_ARGS__)
#else
#define DEBUG_TRACE(S, ...)
#endif
/* Place your own code here */
DEBUG_TRACE("The value is:%d\n",value);

will result in:

printf("The value is:%d\n",value);
```
Descriptions of miscellaneous preprocessor extensions
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.

Introduction

The compiler comes with the IAR DLIB Library, a complete ISO/ANSI C and C++ library. 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, et cetera.

For additional information, 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 additional information about library functions, see the chapter Implementation-defined behavior 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 choose a runtime library, see Basic project configuration, page 19. The linker will include only those routines that are required—directly or indirectly—by your application.
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 these functions and parts are not reentrant because they need static data:

- Heap functions—malloc, free, realloc, calloc, and the C++ operators new and delete
- Time functions—asctime, localtime, gmtime, mktime
- Multibyte functions—mbrlen, mbtowc, mbsrtowc, wcrtomb, wcsrtomb, wctomb
- The miscellaneous functions setlocale, rand, atexit, sterror, strtok
- Functions that use files in some way. This includes printf, scanf, getchar, and putchar. The functions sprintf and sscanf are not included.

Some functions also share the same storage for errno. These functions are not reentrant, since an errno value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. Among these functions are:

- exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2, cosh, sinh, strtod, strtol, strtoul

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.

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 the ISO/ANSI standard for the programming language C. For additional information, see the chapter Implementation-defined behaviour in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- CSTARTUP, the module containing the start-up code. It is described in 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, partly taken
from the C99 standard, see Added C functionality, page 296.

## 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
Compiler extensions.

The following 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>ctype.h</td>
<td>Classifying characters</td>
</tr>
<tr>
<td>errno.h</td>
<td>Testing error codes reported by library functions</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>time.h</td>
<td>Converting between various time and date formats</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 33: Traditional standard C header files—DLIB
## C++ HEADER FILES

This section lists the C++ header files.

### Embedded C++

The following table lists the Embedded C++ header files:

<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</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>istream</td>
<td>Declaring the I/O stream objects that manipulate the standard streams</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</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>strstream</td>
<td>Defining several I/O stream classes that manipulate in-memory character sequences</td>
</tr>
</tbody>
</table>

### Table 34: Embedded C++ header files

The following table lists additional C++ header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>fstream.h</td>
<td>Defining several I/O stream classes that manipulate external files</td>
</tr>
<tr>
<td>iomanip.h</td>
<td>Declaring several I/O stream manipulators that take an argument</td>
</tr>
<tr>
<td>istream.h</td>
<td>Declaring the I/O stream objects that manipulate the standard streams</td>
</tr>
<tr>
<td>new.h</td>
<td>Declaring several functions that allocate and free storage</td>
</tr>
</tbody>
</table>

### Table 35: Additional Embedded C++ header files—DLIB
Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

<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>deque</td>
<td>A deque sequence container</td>
</tr>
<tr>
<td>functional</td>
<td>Defines several function objects</td>
</tr>
<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>vector</td>
<td>A vector sequence container</td>
</tr>
</tbody>
</table>

Table 36: Standard template library header files

Using standard C libraries in C++

The C++ library works in conjunction with 15 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.

The following 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>cctype</td>
<td>Classifying characters</td>
</tr>
<tr>
<td>cerrno</td>
<td>Testing error codes reported by library functions</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>
</tbody>
</table>

Table 37: New standard C header files—DLIB
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, partly taken from the C99 standard.

The following include files provide these features:

- ctype.h
- inttypes.h
- math.h
- stdbool.h
- stdint.h
- stdio.h
- stdlib.h
- wchar.h

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>climits</td>
<td>Testing integer type properties</td>
</tr>
<tr>
<td>locale</td>
<td>Adapting to different cultural conventions</td>
</tr>
<tr>
<td>cmath</td>
<td>Computing common mathematical functions</td>
</tr>
<tr>
<td>csetjmp</td>
<td>Executing non-local goto statements</td>
</tr>
<tr>
<td>csignal</td>
<td>Controlling various exceptional conditions</td>
</tr>
<tr>
<td>cstdarg</td>
<td>Accessing a varying number of arguments</td>
</tr>
<tr>
<td>cstddef</td>
<td>Adds support for the bool data type in C.</td>
</tr>
<tr>
<td>cstddef</td>
<td>Defining several useful types and macros</td>
</tr>
<tr>
<td>cstdarg</td>
<td>Providing integer characteristics</td>
</tr>
<tr>
<td>cstdio</td>
<td>Performing input and output</td>
</tr>
<tr>
<td>cstdlib</td>
<td>Performing a variety of operations</td>
</tr>
<tr>
<td>cstring</td>
<td>Manipulating several kinds of strings</td>
</tr>
<tr>
<td>ctime</td>
<td>Converting between various time and date formats</td>
</tr>
<tr>
<td>cwchar</td>
<td>Support for wide characters</td>
</tr>
<tr>
<td>cwctype</td>
<td>Classifying wide characters</td>
</tr>
</tbody>
</table>

Table 37: New standard C header files—DLIB (Continued)
● wctype.h

ctype.h
In ctype.h, the C99 function isblank is defined.

inttypes.h
This include file defines the formatters for all types defined in stdint.h to be used by the functions printf, scanf, and all their variants.

math.h
In math.h all functions exist in a float variant and a long double variant, suffixed by f and l respectively. For example, sinf and sinl.

The following C99 macro symbols are defined:
HUGE_VALF, HUGE_VALL, INFINITY, NAN, FP_INFINITE, FP_NAN, FP_NORMAL,
FP_SUBNORMAL, FP_ZERO, MATH_ERRNO, MATH_ERREXCEPT, math_errhandling.

The following C99 macro functions are defined:
fpclassify, signbit, isfinite, isnan, isnormal, isgreater, isless, islessequal, islessgreater, isunordered.

The following C99 type definitions are added:
float_t, double_t.

stdbool.h
This include file makes the bool type available if the Allow IAR extensions (-e) option is used.

stdint.h
This include file provides integer characteristics.

stdio.h
In stdio.h, the following C99 functions are defined:
vscanf, vfscanf, vsscanf, vsnprintf, snprintf

The functions printf, scanf, and all their variants have added functionality from the C99 standard. For reference information about these functions, see the library reference available from the Help menu.
The following functions providing I/O functionality for libraries built without FILE support are defined:

- **__write_array** Corresponds to fwrite on stdout.
- **__ungetchar** Corresponds to ungetc on stdout.
- **__gets** Corresponds to fgets on stdin.

### stdlib.h

In stdlib.h, the following C99 functions are defined:

- _Exit, llabs, lldiv, strtoll, strtoull, atol, strtof, strtold.

The function strtod has added functionality from the C99 standard. For reference information about this function, see the library reference available from the Help menu.

The __qsortbbl function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

### wchar.h

In wchar.h, the following C99 functions are defined:

- vfwscanf, vswscanf, vwscanf, wcstof, wcstolb.

### wctype.h

In wctype.h, the C99 function iswblank is defined.
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 40 IAR C/C++ Development Guide for ARM®.

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
  meaning that you can include or exclude directives depending on a condition, and to
  split the configuration file into several different files.

Comments can be written either as C comments (/*...*/) or as C++ comments
(/.../).

Defining memories and regions

ILINK needs information about the available memory spaces, or more specifically it
needs information about:

- The maximum size of possible addressable memories
  The define memory directive defines a memory space with a given size, which is
  the maximum possible amount of addressable memory, not necessarily physically
  available. See Define memory directive, page 300.

- 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 301.

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 303.

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 317.

**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.
The linker configuration file

**Define memory directive**

**Description**

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**

```c
/* Declare the memory space Mem of four Gigabytes */
define memory Mem with size = 4G;
```

**Define region directive**

**Syntax**

```c
define region name = region-expr;
```

where `region-expr` is a region expression, see also Regions, page 301.

**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**

```c
/* 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**

```c
[ memory-name:] [from expr { to expr | size expr }]
[ repeat expr [ displacement expr ]]}
```
where \( expr \) is an expression, see \textit{Expressions}, page 317.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{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>\textit{from}</td>
<td>The start address of the memory range (inclusive).</td>
</tr>
<tr>
<td>\textit{to}</td>
<td>The end address of the memory range (inclusive).</td>
</tr>
<tr>
<td>\textit{size}</td>
<td>The size of the memory range.</td>
</tr>
<tr>
<td>\textit{repeat}</td>
<td>Defines several ranges in the same memory for the region literal. Defaults to 1.</td>
</tr>
<tr>
<td>\textit{displacement}</td>
<td>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 the range can even be expressed by unsigned values, because it is known where the memory wraps.

The \textit{repeat} parameter will create a region literal that contains several ranges, one for each repeat. This is useful for \textit{banked} or \textit{far} regions.

### Example

/* The 5-byte size range spans over the address zero */
Mem:[from -2 to 2]

/* The 512-byte size range spans over zero, in a 64-Kbyte memory */
Mem:[from 0xFF00 to 0xFF]

/* Defining several ranges in the same memory, a repeating literal */
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]

/* Resulting in a region containing: */
  Mem:[from 0 size 0x100]
  Mem:[from 0x1000 size 0x100]
  Mem:[from 0x2000 size 0x100]

### See also

\textit{Define region directive}, page 301, and \textit{Region expression}, page 303.
Region expression

Syntax

\[
\text{region-operand} \\
| \text{region-expr} | \text{region-operand} \\
| \text{region-expr} - \text{region-operand} \\
| \text{region-expr} & \text{region-operand}
\]

where \text{region-operand} is one of:

\[
\{ \text{region-expr} \} \\
\text{region-name} \\
\text{region-literal} \\
\text{empty-region}
\]

where \text{region-name} is a region, see Define region directive, page 301
where \text{region-literal} is a region literal, see Region literal, page 301
and where \text{empty-region} is an empty region, see Empty region, page 304.

Description

Normally, a region consists of one memory range, which means a region literal is sufficient to express it. When a region contains several ranges, possibly in different memories, it is instead necessary to use a region expression to express it. Region expressions are actually set expressions on sets of memory ranges.

To create region expressions, three operators are available: union (|), intersection (&), and difference (\(-\)). These operators work as in set theory. For example, if you have the sets A and B, then the result of the operators would be:

- \( A \ | \ B \): all elements in either set A or set B
- \( A \ & \ B \): all elements in both set A and B
- \( A \ - \ B \): all elements in set A but not in B.

Example

/* Resulting in a range starting at 1000 and ending at 2FFF, in memory Mem */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1500 and ending at 1FFF, in memory Mem */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1000 and ending at 14FF, in memory Mem */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]
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. */

See also Region expression, page 303.

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 311 and Place in directive, page 312.

- 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 305, and Define overlay directive, page 306.
 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 307 and Do not initialize directive, page 310.

 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 310.

Define block directive

Syntax
define [movable] block name
   [with param, param... ]
   {
      extended-selectors
   }
   [except
      {
         section_selectors
      }]
;

where param can be one of:
size = expr
maximum size = expr
alignment = expr
fixed order
static base

and where the rest of the directive selects sections to include in the block, see Section selection, page 312.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the defined block.</td>
</tr>
<tr>
<td>size</td>
<td>Customizes the size of the block. By default, the size of a block is the sum of its parts dependent of its contents.</td>
</tr>
<tr>
<td>maximum size</td>
<td>Specifies an upper limit for the size of the block. An error is generated if the sections in the block do not fit.</td>
</tr>
<tr>
<td>alignment</td>
<td>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.</td>
</tr>
</tbody>
</table>
Section handling

**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 { };

**See also**

Interaction between the tools and your application, page 115. See Define overlay directive, page 306 for an accessing example.

### Define overlay directive

**Syntax**

```c
define overlay name [ with param, param... ]
{ extended-selectors;
} [except
 { section_selectors
 }];
```

For information about extended selectors and except clauses, see Section selection, page 312.

**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.
The linker configuration file

**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 53
*IAR C/C++ Development Guide for ARM®* for examples of using the `define overlay` directive.

**Initialize directive**

**Syntax**

```plaintext
initialize { by copy | manually }  
  [ with param, param... ]  
  {  
    section-selectors  
  }  
{  
  except  
  {  
    section_selectors  
  }  };
```

**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.
where `param` is one of:

\[
\text{packing} = \{ \text{none} | \text{zeros} | \text{packbits} | \text{bwt} | \text{lzw} | \text{auto} | \text{smallest} \}
\]

and where the rest of the directive selects sections to include in the block. See Section selection, page 312.

### Parameters

**by copy**

Splits the section into sections for initializers and initialized data, and handles the initialization at application startup automatically.

**manually**

Splits the section into sections for initializers and initialized data. The initialization at application startup is not handled automatically.

**packing**

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 sequential 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.
- `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` and `packbits`. 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 zeros and packbits. Approximately 9 Kbytes of stack space is needed for bwt and 3.5 Kbytes for lzw.

When initializers are compressed, the exact size of the compressed initializers is unknown until the exact content of the uncompressed data is known. If this data contains other addresses, and some of these addresses are dependent on the size of the compressed initializers, the linker fails with error Lp017. To avoid this, place compressed initializers last, or in a memory region together with sections whose addresses do not need to be known.

Unless initialize manually is used, ILINK will arrange for initialization to occur during system startup by including an initialization table. Startup code calls an initialization routine that reads this table and performs the necessary initializations.

Zero-initialized sections are not affected by the initialize directive.

The initialize directive is normally used for initialized variables, but can be used for copying any sections, for example copying executable code from slow ROM to fast RAM, or for overlays. For another example, see Define overlay directive, page 306.

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 45, and Do not initialize directive, page 310.
**Do not initialize directive**

**Syntax**

```c
do not initialize
{
    section-selectors
}
[except
{
    section-selectors
}]
;
```

For information about extended selectors and except clauses, see *Section selection*, page 312.

**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 45, and *Initialize directive*, page 307.

**Keep directive**

**Syntax**

```c
keep
{
    section-selectors
}
[except
{
    section-selectors
}]
;
```

For information about extended selectors and except clauses, see *Section selection*, page 312.

**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**

```c
keep { section .keep* } except {section .keep};
```
The linker configuration file

**Place at directive**

**Syntax**

```plaintext
place at { address [ memory: ] expr | start of region_expr | end of region_expr |
  { extended-selectors }
} [ except { section-selectors } ];
```

For information about extended selectors and except clauses, see Section selection, page 312.

**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. The start of the region is used.

- **end of region_expr**
  A region expression. The end of the region is used.

**Description**

The `place at` directive places sections and blocks either at a specific address or, at the beginning or the end of a region. The same address cannot be used for two different `place at` directives. It is also not possible to use an empty region in a `place at` directive. If placed in a region, the sections and blocks will be placed before any other sections or blocks placed in the same region with a `place in` directive.

The sections and blocks will be placed in the region in an arbitrary order. To specify a specific order, use the `block` directive.

The `name`, if specified, is used in the map file and in some log messages.

**Example**

```plaintext
/* 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 312.
Section selection

Place in directive

Syntax

```c
[ "name": ]
place in region-expr
{
    extended-selectors
}
[except{
    section-selectors
}];
```

where `region-expr` is a region expression, see also `Regions`, page 301.
and where the rest of the directive selects sections to include in the block. See `Section selection`, page 312.

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

```c
/* Place the read-only sections in the code_region */
"ROM": place in ROM { readonly };
```

See also

`Place at directive`, page 311.

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

\[
\{ \text{section-selector} | , \text{section-selector}... \} \\
\text{where section-selector is:}
\]

\[
\{ \text{section-attribute} | \text{section-type} | \text{section sectionname} \}
\]

\text{where section-attribute is:}

\[
\{ \text{ro} | \text{readonly} | \text{rw} | \text{readwrite} | \text{zi} | \text{zeroinit} \}
\]

\text{and where ro, rw, and zi also can be readonly, readwrite, and zeroinit, respectively.}

\text{And section-type is:}

\[
\{ \text{preinit_array} | \text{init_array} \}
\]

Parameters

- **ro or readonly**: Read-only sections.
- **rw or readwrite**: Read/write sections.
- **zi or zeroinit**: Zero-initialized sections. These sections have no content and should possibly be initialized with zeros during system startup.
- **code**: Sections that contain code.
- **data**: Sections that contain data.
- **preinit_array**: Sections of the ELF section type SHT_PREINIT_ARRAY.
- **init_array**: Sections of the ELF section type SHT_INIT_ARRAY.
- **sectionname**: The section name. Two wildcards are allowed:

  - `?` matches any single character
  - `*` matches zero or more characters.

- **module**: A name in the form `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.

Description

A section selector selects all sections that match the section attribute, section type, section name, and the name of the object, where object is an object file, a library, or an
object in a library. Up to three of the four conditions can be omitted. If the section attribute is omitted, any section will be selected, without restrictions on the section attribute. If the section type is omitted, sections of any type will be selected.

If the section name part or the object name part is omitted, sections will be selected without restrictions on the section name or object name, respectively.

It is also possible to use only {} without any section selectors, which can be useful when defining blocks.

Note that a section selector with narrower scope has higher priority than a more generic section selector.

If more than one section selector matches for the same purpose, one of them must be more specific. A section selector is more specific than another one if:

- It specifies a section type and the other one does not
- It specifies a section name or object name with no wildcards and the other one does not
- There could be sections that match the other selector that also match this one, and the reverse is not true.

<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>ro 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 code section &quot;*x&quot;</td>
<td>Selector 1</td>
</tr>
<tr>
<td>section &quot;.intvec&quot;</td>
<td>ro code 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 38: Examples of section selector specifications

Example

```c
{ 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:

```c
object foo.o(lib.a)
object f*(lib*)
object foo.o
object lib.a
```
See also Initialize directive, page 307, Do not initialize directive, page 310, and Keep directive, page 310.

Extended-selectors

Syntax

\[
\{ \ [ \ extended-selector \ ]\ [ , \ extended-selector \ldots ] \}
\]

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

```plaintext
define block First { section .first }; /* Define a block holding the section .first */
define block Table { first block First }; /* Define a block where the first is placed first */
```

or, equivalently using an inline definition of the block First:

```plaintext
define block Table { first block First { section .first }};
```

See also Define block directive, page 305, Define overlay directive, page 306, and Place at directive, page 311.
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 316, and `Export directive`, page 317.

- Use expressions and numbers
  In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, etc. See `Expressions`, page 317.

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 317 and `Interaction between ILINK and the application`, page 55.
Export directive

Syntax

```plaintext
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.

Example

```plaintext
/* Define the symbol my_symbol to be exported */
export symbol my_symbol;
```

Expressions

Syntax

An expression is built up of the following constituents:

- `expression` is an expression.
- `binop` is one of these binary operators:
  - `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `>`, `==`, `!=`, `&`, `^`, `|`, `&&`, `||`
- `unop` is one of these unary operators:
  - `+`, `-`, `!`, `~`
- `number` is a number, see `Numbers`, page 318
- `symbol` is a defined symbol, see `Define symbol directive`, page 316 and `--config_def`, page 192
- `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.

Expressions

```plaintext
evaluation result
```
Using symbols, expressions, and numbers

isDefinedSymbol(expr-symbol) Returns True if the expression symbol is defined, otherwise False.

start(r) Returns the lowest address in the region.

dend(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 303.

Description

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-suffix]

where nr is either a decimal number or a hexadecimal number (0x... or 0X...).

and where nr-suffix is one of:

K /* Kilo = (1 << 10) 1024 */
M /* Mega = (1 << 20) 1048576 */
G /* Giga = (1 << 30) 1073741824 */
T /* Tera = (1 << 40) 1099511627776 */
P /* Peta = (1 << 50) 1125899906842624 */

Description

A number can be expressed either by normal C means or by suffixing it with a set of useful suffixes, which provides a compact way of specifying numbers.

Example

1024 is the same as 0x400, which is the same as 1K.
Structural configuration

The structural directives provide means for creating structure within the configuration, such as:

- Conditional inclusion
  An if directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations in the same file. See If directive, page 319.
- Dividing the linker configuration file into several different files
  The include directive makes it possible to divide the configuration file into several logically distinct files. See Include directive, page 320.

If directive

Syntax

```plaintext
if (expr) {
  directives
} else if (expr) {
  directives
} else {
  directives
}
```

where `expr` is an expression, see Expressions, page 317.

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 304.
### 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 files. For instance, files that you need to change often and files that you seldom edit.
Section reference

The compiler places code and data into sections. Based on a configuration specified in the linker configuration file, ILINK places sections in memory.

This chapter lists all predefined 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 40.

Summary of sections

This table lists the 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 C or C++ 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, including the initializers.</td>
</tr>
<tr>
<td>.data_init</td>
<td>Holds initializers for .data sections.</td>
</tr>
<tr>
<td>.difunct</td>
<td>Holds pointers to code, typically C++ constructors, that should be</td>
</tr>
<tr>
<td></td>
<td>executed by the system startup code before main is called.</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>.intvec</td>
<td>Holds the reset and interrupt vectors.</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>.rodata</td>
<td>Holds constant data.</td>
</tr>
<tr>
<td>.text</td>
<td>Holds the program code.</td>
</tr>
</tbody>
</table>

Table 39: 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 the *Placing code and data—the linker configuration file*, page 42.

---

**.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 52.
.cstart
Description: Holds the startup code.
Memory placement: This section can be placed anywhere in memory.

.data
Description: Holds static and global initialized variables including initializers.
Memory placement: This section can be placed anywhere in memory.

.data_init
Description: Holds initializers for .data sections. This section is created by the linker.
Memory placement: This section can be placed anywhere in memory.

.difunct
Description: Holds the dynamic initialization vector used by C++.
Memory placement: This section can be placed anywhere in memory.

HEAP
Description: Holds the heap used for dynamically allocated data, in other words data allocated by malloc and free, and in C++, new and delete.
Memory placement: This section can be placed anywhere in memory.
See also: Setting up the heap, page 52.

. iar.dynexit
Description: Holds the table of calls to be made at exit.
Memory placement: This section can be placed anywhere in memory.
Descriptions of sections and blocks

See also Setting up the atexit limit, page 52.

.intvec
Description Holds the reset vector and exception vectors which contain branch instructions to cstartup, interrupt service routines etc.
Memory placement Must be placed at address range 0x00 to 0x3F.

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 stacks, page 114.

.noinit
Description Holds static and global __no_init variables.
Memory placement This section can be placed anywhere in memory.

.rodata
Description Holds constant data. This can include constant variables, string and aggregate literals, etc.
Memory placement This section can be placed anywhere in memory.

.text
Description Holds program code, except the code for system initialization.
Memory placement This section can be placed anywhere in memory.
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.

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 IAR Embedded Workbench® IDE User Guide for ARM®.

INVOCATION SYNTAX

The invocation syntax for the archive builder is:

iarchive parameters
The IAR Archive Tool—iarchive

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</td>
<td>The object file(s) that the specified command operates on.</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>objectfileN</td>
<td></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 40: 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 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 41: iarchive commands summary

For more detailed descriptions, see Descriptions of command line options, page 327.
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>-o</code></td>
<td>Specifies the library file.</td>
</tr>
<tr>
<td><code>--silent</code>, <code>-S</code></td>
<td>Sets silent operation.</td>
</tr>
<tr>
<td><code>--verbose</code>, <code>-V</code></td>
<td>Reports all performed operations.</td>
</tr>
</tbody>
</table>

Table 42: Iarchive options summary

DESCRIPTIONS OF COMMAND LINE OPTIONS
This section gives detailed reference information about each `iarchive` command line option.

```
-f
```

Syntax

```
-f filename
```

Parameters

For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 154.

Description

Use this option to make `iarchive` 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.

```
--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 154.
The IAR Archive Tool—iarchive

objectfile1 ... The object file(s) to build the library from.
objectfileN

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.

--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 154.

objectfile1 ... The object file(s) that the command operates on.
objectfileN

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.

--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 154.

objectfile1 ... The object file(s) that the command operates on.
objectfileN
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.

-o
Syntax
-o libraryfile
Parameters
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 154.
Description
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.

--replace, -r
Syntax
--replace libraryfile objectfile1 ... objectfileN
-r 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 154.
objectfile1 ... The object file(s) that the command operates on.
objectfileN
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.

--silent, -S
Syntax
--silent
-S
Description
Use this option to make iarchive operate without sending any messages to the standard output stream.
By default, `iarchive` sends various messages via the standard output stream. You can use this option to prevent this. `iarchive` sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

`--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 154.

**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.

`--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 154.

**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. In silent mode (`--silent`), this command performs symbol table-related syntax checks on the library file and displays only errors and warnings.

`--verbose, -V`

**Syntax**
```
--verbose
-V
```

**Description**
Use this option to make `iarchive` report which operations it performs, in addition to giving diagnostic messages.
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>inputfile</td>
<td>An absolute ELF executable image produced by the ILINK linker.</td>
</tr>
</tbody>
</table>

Table 43: ielftool parameters
For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 154.

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 section gives detailed reference information about each ielftool option.

**DESCRIPTIONS OF OPTIONS**

This section gives detailed reference information about each ielftool option.
--bin

Syntax

--bin

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|r[i|p]

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

1|2

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.

r

Reverses the byte order of the input data within each word of size size.
Use either \texttt{i} or \texttt{p}, if the \texttt{start} value is bigger than 0. If specified, \texttt{p} can be used for prefixing.
- \texttt{i} - Initializes the checksum value with the start value.
- \texttt{p} - Prefixes the input data with a word of size \texttt{size} that contains the start value.

\texttt{start} By default, the initial value of the checksum is 0. If necessary, use \texttt{start} to supply a different initial value. If not 0, then either \texttt{i} or \texttt{p} must be specified.

\texttt{range} The address range on which the checksum should be calculated. Hexadecimal and decimal notation is allowed (for example, 0x8002–0x8FFF).

\section*{Description}
Use this option to calculate a checksum with the specified algorithm for the specified ranges. The checksum will then replace the original value in \texttt{symbol}. A new absolute symbol will be generated; with the \texttt{symbol} name suffixed with \texttt{._value} containing the calculated checksum. This symbol can be used for accessing the checksum value later when needed, for example during debugging.

If the \texttt{--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 \texttt{symbol} that is specified in a later evaluated \texttt{--checksum} option, an error is issued.

\section*{Example}
This example shows how to use the \texttt{crc16} algorithm with the start value 0 over the address range 0x8000–0x8FFF:
\begin{verbatim}
ielftool --checksum=__checksum:2,crc16;0x8000-0x8FFF sourceFile.out destinationFile.out
\end{verbatim}
The input data is read from \texttt{sourceFile.out}, and the resulting checksum value of size 2 bytes will be stored at the symbol \texttt{__checksum}. The modified ELF file is saved as \texttt{destinationFile.out} leaving \texttt{sourceFile.out} untouched.

\section*{See also}
Checksum calculation, page 117

To set related options, choose:

\begin{verbatim}
Project>Options>Linker>Checksum
\end{verbatim}
--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.

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

Description

Sets the format of the output file to linear Intel hex.

To set related options, choose:

Project>Options>Linker>Output converter
--silent

Syntax

--silent

Description

Causes ielftool 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

Syntax

--simple

Description

Sets the format of the output file to Simple code.

To set related options, choose:

Project>Options>Output converter

--srec

Syntax

--srec

Description

Sets the format of the output file to Motorola S-records.

To set related options, choose:

Project>Options>Output converter

--srec-len

Syntax

--srec-len=length

Parameters

length The number of data bytes in each S-record.
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.

Description
Restricts the number of data bytes in each S-record. This option can be used in combination with the --srec option.

This option is not available in the IDE.

--srec-s3only

Syntax
--srec-s3only

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

Description
Removes debug information from the ELF output file. 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

--verbose

Syntax
--verbose

Description
Use this option to make ielftool 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.
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.

**INVOCACTION SYNTAX**

The invocation syntax for ielfdumparm is:

```
ielfdumparm filename
```

**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><code>filename</code></td>
<td>An ELF relocatable or executable file to use as input.</td>
</tr>
</tbody>
</table>

Table 45: ielfdumparm parameters

For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 154.

**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><code>--all</code></td>
<td>Generates output for all input sections regardless of their names or numbers.</td>
</tr>
<tr>
<td><code>-o</code></td>
<td>Specifies an output file.</td>
</tr>
<tr>
<td><code>--raw</code></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><code>--section/-s</code></td>
<td>Generates output for selected input sections.</td>
</tr>
</tbody>
</table>

Table 46: ielfdumparm options summary
DESCRIPTIONS OF OPTIONS

This section gives detailed reference information about each ielfdumparm option.

--all

Syntax

--all

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.

-o, --output

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 154.

Description

By default, output from the dumper is directed to the console. Use this option to direct the output to a file instead.

If you specify a directory, the output file will be named the same as the input file, only with an extra id extension.

--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.

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

```plaintext
-s 3,17                  /* Sections #3 and #17
-s .debug_frame,42       /* Any sections named .debug_frame and also section #42 */
```

--raw

Syntax

```plaintext
--raw
```

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.

---

**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:

```plaintext
iobjmanip options inputfile outputfile
```

**Parameters**

The parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>options</strong></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><strong>inputfile</strong></td>
<td>A relocatable ELF object file.</td>
</tr>
</tbody>
</table>

*Table 47: iobjmanip parameters*
For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 154.

**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><code>-f</code></td>
<td>Extends the command line.</td>
</tr>
<tr>
<td><code>--remove_section</code></td>
<td>Removes a section.</td>
</tr>
<tr>
<td><code>--rename_section</code></td>
<td>Renames a section.</td>
</tr>
<tr>
<td><code>--rename_symbol</code></td>
<td>Renames a symbol.</td>
</tr>
<tr>
<td><code>--strip</code></td>
<td>Removes debug information.</td>
</tr>
</tbody>
</table>

**DESCRIPTIONS OF COMMAND LINE OPTIONS**

This section gives detailed reference information about each `iobjmanip` command line option.

**-f**

**Syntax**

```
-f filename
```

**Parameters**

For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 154.

**Description**

Use this option to make `iobjmanip` 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.

**--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`.

Description

Use this option to make `iobjmanip` omit the specified section when generating the output file.

**--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.

Description

Use this option to make `iobjmanip` rename the specified section when generating the output file.

**--rename_symbol**

Syntax

```
--rename_symbol oldname =newname
```

Parameters

- **oldname**: The symbol to be renamed.
- **newname**: The new name of the symbol.
Description
Use this option to make \texttt{iobjmanip} rename the specified symbol when generating the output file.

\textbf{--strip}

Syntax
\texttt{--strip}

Description
Use this option to remove all sections containing debug information before writing the output file.

To set related options, choose:
\texttt{Project>Options>Linker>Output>Include debug information in output}

\textbf{DIAGNOSTIC MESSAGES}

This section lists the messages produced by \texttt{iobjmanip}:

\textbf{Lm001: No operation given}
None of the command line parameters specified an operation to perform.

\textbf{Lm002: Expected \texttt{nr1} parameters but got \texttt{nr2}}
Too few or too many parameters. Check invocation syntax for \texttt{iobjmanip} and for the used command line options.

\textbf{Lm003: Invalid section/symbol renaming pattern \textit{pattern}}
The pattern does not define a valid renaming operation.

\textbf{Lm004: Could not open file \textit{filename}}
\texttt{iobjmanip} failed to open the input file.

\textbf{Lm005: ELF format error \textit{msg}}
The input file is not a valid ELF object file.

\textbf{Lm006: Unsupported section type \texttt{nr}}
The object file contains a section that \texttt{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 name1 removed due to transitive dependency on name2
A section was removed as it depends on an explicitly removed section.

Lm014: File has no section with index nr
A section index, used as a parameter to --remove_section or --rename_section, did not refer to a section in the input file.

Ms003: could not open file filename for writing
iobjmanip failed to open the output file for writing. Make sure that it is not write protected.

Ms004: problem writing to file filename
An error occurred while writing to file filename. A possible reason for this is that the volume is full.
Ms005: problem closing file filename
An error occurred while closing the file filename.

The IAR Absolute Symbol Exporter—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:

\texttt{isymexport [options] inputfile outputfile [options]}

Parameters

The parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputfile</td>
<td>A ROM image in the form of an executable ELF file (output from linking).</td>
</tr>
<tr>
<td>options</td>
<td>Any of the available command line options, see Summary of isymexport options, page 347.</td>
</tr>
<tr>
<td>outputfile</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>

For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 154.
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; for more information, see -f, page 169.</td>
</tr>
</tbody>
</table>

Table 50: isymexport options summary

DESCRIPTIONS OF OPTIONS

This section gives detailed reference information about each isymexport option.

--edit

Syntax

--edit steering_file

Description

Use this option to specify a steering file to control what 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 347.

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 zzza, but not zzzaaa */
hide zzzx /* But do not export zzzx */

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.
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.
The IAR Absolute Symbol Exporter—isymexport
Implementation-defined behavior

This chapter describes how the compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - Programming Languages - C (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, Technical Corrigendum 1, and Technical Corrigendum 2, contains an appendix called Portability Issues. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

Translation

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

`filename,linenumber level[tag]: message`

where `filename` is the name of the source file in which the error was encountered, `linenumber` is the line number at which the compiler detected the error, `level` is the level of seriousness of the message (remark, warning, error, or fatal error), `tag` is a unique tag that identifies the message, and `message` is an explanatory message, possibly several lines.
Environment

Arguments to main (5.1.2.2.1)
The function called at program startup is called main. No prototype was declared for
main, and the only definition supported for main is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see Customizing
system initialization, page 74.

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 79.

**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 79.

**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 210, 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 213, 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 215, for information about size_t.

Conversion from/to pointers (6.3.4)
See Casting, page 215, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)
See ptrdiff_t, page 216, for information about the ptrdiff_t.

Registers

Honoring the register keyword (6.5.1)
User requests for register variables are not honored.

Structures, unions, enumerations, and bitfields

Improper access to a union (6.3.2.3)
If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

Padding and alignment of structure members (6.5.2.1)
See the section Basic data types, page 210, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)
A 'plain' int bitfield is treated as a unsigned int bitfield. All integer types are allowed as bitfields.
Allocation order of bitfields within a unit (6.5.2.1)
Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)
Bitfields can 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:
```c
#include "mydirectory\myfile"
```
Within source code, two backslashes are necessary:
```c
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:
```c
alignment
baseaddr
basic_template_matching
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
```
Descriptions of implementation-defined behavior

```
keep_definition
memory
module_name
no_pch
once
__printf_args
public_equ
__scanf_args
STDC
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:lineno 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 82.

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 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 78.

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 78.

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 78.
%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 81.
system() (7.10.4.5)
How the command processor works depends on how you have implemented the `system` function. See Environment interaction, page 81.

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></td>
</tr>
<tr>
<td>all others</td>
<td>error nnn</td>
</tr>
</tbody>
</table>

Table 51: 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 83.

clock() (7.12.2.1)
From where the system clock starts counting depends on how you have implemented the `clock` function. See Time, page 83.
Descriptions of implementation-defined behavior
Index

A

--aapcs (compiler option) ......................... 159
ABI, AEABI and IA64 ............................. 121
abort
   implementation-defined behavior (DLIB) .......... 362
   system termination (DLIB) ....................... 73
Abort_Handler (exception function) ................ 32
__absolute (extended keyword) .................... 237
absolute location
   data, placing at (@) ............................ 130
   language support for ........................... 222
   #pragma location ................................ 251
--aeabi (compiler option) ................... 160
_AEABI_PORTABILITY_LEVEL (preprocessor symbol) 123
_AEABI_PORTABLE (preprocessor symbol) .......... 123
algorithm (STL header file) ...................... 295
alignment
   forcing stricter (#pragma data_alignment) ...... 247
   in structures (#pragma pack) ................. 254
   in structures, causing problems ............... 127
   of an object (__ALIGNOF__) ..................... 223
   of data types .................................. 209
   restrictions for inline assembler .............. 92
alignment (pragma directive) .................... 359
__ALIGNOF__ (operator) .......................... 223
--align_sp_on_irq (compiler option) ............. 160
--all (elfdump option) .......................... 340
anonymous structures ............................ 127
anonymous symbols, creating ..................... 226
application
   execution, overview of .......................... 14
   startup and termination (DLIB) ................. 70
architecture
   more information about ........................ xxix
   of ARM ........................................ 25
ARM
   and Thumb code, overview ........................ 29
CPU mode ........................................ 20
memory layout .................................... 25
   supported devices ................................ 5
__arm (extended keyword) ....................... 237
--arm (compiler option) ........................ 160
__ARMVFP__ (predefined symbol) .................. 284
__ARMVFPV1__ (predefined symbol) ............... 284
__ARMVFPV2__ (predefined symbol) ............... 284
__ARMVFPV3__ (predefined symbol) ............... 284
__ARMVFPV4__ (predefined symbol) ............... 284
__ARMVFP_D6__ (predefined symbol) .............. 284
__ARMVFP_FP16__ (predefined symbol) .......... 284
__ARMVFP_SP__ (predefined symbol) .............. 284
__ARM_PROFILE_M__ (predefined symbol) ......... 284
__ARM4M__ (predefined symbol) ................... 284
__ARM4TM__ (predefined symbol) .................. 284
__ARM5__ (predefined symbol) .................... 284
__ARM5E__ (predefined symbol) ................... 284
__ARM6__ (predefined symbol) .................... 284
__ARM6M__ (predefined symbol) ................... 284
__ARM6SM__ (predefined symbol) .................. 284
__ARM7EM__ (predefined symbol) .................. 284
__ARM7M__ (predefined symbol) ................... 284
__ARM7R__ (predefined symbol) ................... 284
arrays
   designated initializers in ........................ 227
   implementation-defined behavior ................. 357
   incomplete at end of structs ................... 226
   non-lvalue .................................... 230
   of incomplete types ................................ 228
   single-value initialization ..................... 230
asm, __asm (language extension) ................ 225
assembler code
   calling from C .................................. 92
   calling from C++ ................................ 94
   inserting inline ................................ 91
assembler directives
   for call frame information ..................... 101
   using in inline assembler code .................. 92
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>calloc (library function)</td>
<td>27</td>
</tr>
<tr>
<td>See also</td>
<td></td>
</tr>
<tr>
<td>implementation-defined behavior of (DLIB)</td>
<td>362</td>
</tr>
<tr>
<td>canInstantiate (pragma directive)</td>
<td>359</td>
</tr>
<tr>
<td>cast (DLIB header file)</td>
<td>295</td>
</tr>
<tr>
<td>cast operators</td>
<td></td>
</tr>
<tr>
<td>in Extended EC++</td>
<td>106</td>
</tr>
<tr>
<td>missing from Embedded C++</td>
<td>106</td>
</tr>
<tr>
<td>casting</td>
<td></td>
</tr>
<tr>
<td>of pointers and integers</td>
<td>215</td>
</tr>
<tr>
<td>pointers to integers, language extension</td>
<td>229</td>
</tr>
<tr>
<td>cctype (DLIB header file)</td>
<td>295</td>
</tr>
<tr>
<td>cermo (DLIB header file)</td>
<td>295</td>
</tr>
<tr>
<td>cexit (system termination code)</td>
<td>70</td>
</tr>
<tr>
<td>CFI (assembler directive)</td>
<td>101</td>
</tr>
<tr>
<td>cfiCommon.i (CFI header example file)</td>
<td>104</td>
</tr>
<tr>
<td>CFI_COMMON_ARM (CFI macro)</td>
<td>104</td>
</tr>
<tr>
<td>CFI_COMMON_Thumb (CFI macro)</td>
<td>104</td>
</tr>
<tr>
<td>CFI_NAME_BLOCK (CFI macro)</td>
<td>104</td>
</tr>
<tr>
<td>cfloat (DLIB header file)</td>
<td>295</td>
</tr>
<tr>
<td>char (data type)</td>
<td>210</td>
</tr>
<tr>
<td>changing default representation (~char_is_signed)</td>
<td>160</td>
</tr>
<tr>
<td>signed and unsigned</td>
<td>211</td>
</tr>
<tr>
<td>characters, implementation-defined behavior of</td>
<td>354</td>
</tr>
<tr>
<td>character-based I/O</td>
<td></td>
</tr>
<tr>
<td>in DLIB</td>
<td>75</td>
</tr>
<tr>
<td>overriding in runtime library</td>
<td>67</td>
</tr>
<tr>
<td>~char_is_signed (compiler option)</td>
<td>160</td>
</tr>
<tr>
<td>checksum</td>
<td></td>
</tr>
<tr>
<td>calculation of</td>
<td>117</td>
</tr>
<tr>
<td>display format in C-SPY for symbol</td>
<td>120</td>
</tr>
<tr>
<td>~checksum (ielftool option)</td>
<td>334</td>
</tr>
<tr>
<td>cinttypes (DLIB header file)</td>
<td>295</td>
</tr>
<tr>
<td>classes</td>
<td>107</td>
</tr>
<tr>
<td>cmits (DLIB header file)</td>
<td>296</td>
</tr>
<tr>
<td>dolcail (DLIB header file)</td>
<td>296</td>
</tr>
<tr>
<td>clock (DLIB library function), implementation-defined behavior of</td>
<td>363</td>
</tr>
<tr>
<td>clock.c</td>
<td>83</td>
</tr>
<tr>
<td>__close (DLIB library function)</td>
<td>79</td>
</tr>
<tr>
<td>clustering (compiler transformation)</td>
<td>136</td>
</tr>
<tr>
<td>disabling (~no_clustering)</td>
<td>174</td>
</tr>
<tr>
<td>__CLZ (intrinsic function)</td>
<td>262</td>
</tr>
<tr>
<td>cmath (system initialization code)</td>
<td>70</td>
</tr>
<tr>
<td>cmath (DLIB header file)</td>
<td>296</td>
</tr>
<tr>
<td>code</td>
<td></td>
</tr>
<tr>
<td>ARM and Thumb, overview</td>
<td>29</td>
</tr>
<tr>
<td>interruption of execution</td>
<td>31</td>
</tr>
<tr>
<td>code motion (compiler transformation)</td>
<td>135</td>
</tr>
<tr>
<td>disabling (~no_code_motion)</td>
<td>174</td>
</tr>
<tr>
<td>command line options</td>
<td></td>
</tr>
<tr>
<td>part of compiler invocation syntax</td>
<td>145</td>
</tr>
<tr>
<td>part of linker invocation syntax</td>
<td>145</td>
</tr>
<tr>
<td>passing</td>
<td>146</td>
</tr>
<tr>
<td>See also compiler options</td>
<td></td>
</tr>
<tr>
<td>See also linker options</td>
<td></td>
</tr>
<tr>
<td>typographic convention</td>
<td>xxxiii</td>
</tr>
<tr>
<td>command prompt icon, in this guide</td>
<td>xxxiii</td>
</tr>
<tr>
<td>comments</td>
<td></td>
</tr>
<tr>
<td>after preprocessor directives</td>
<td>230</td>
</tr>
<tr>
<td>C++ style, using in C code</td>
<td>225</td>
</tr>
<tr>
<td>common block (call frame information)</td>
<td>101</td>
</tr>
<tr>
<td>common subexpr elimination (compiler transformation)</td>
<td>135</td>
</tr>
<tr>
<td>disabling (~no_cse)</td>
<td>175</td>
</tr>
<tr>
<td>compilation date</td>
<td></td>
</tr>
<tr>
<td>exact time of (~<em>TIME</em>)</td>
<td>286</td>
</tr>
<tr>
<td>identifying (~<em>DATE</em>)</td>
<td>285</td>
</tr>
<tr>
<td>compiler</td>
<td></td>
</tr>
<tr>
<td>environment variables</td>
<td>146</td>
</tr>
<tr>
<td>invocation syntax</td>
<td>145</td>
</tr>
<tr>
<td>output from</td>
<td>148</td>
</tr>
<tr>
<td>compiler listing, generating (-l)</td>
<td>172</td>
</tr>
<tr>
<td>compiler object file</td>
<td>12</td>
</tr>
<tr>
<td>including debug information in (~debug, -r)</td>
<td>163</td>
</tr>
<tr>
<td>output from compiler</td>
<td>148</td>
</tr>
<tr>
<td>compiler optimization levels</td>
<td>133</td>
</tr>
</tbody>
</table>
ctype.h (library header file) ........................................ 293
data types ............................................................ 210
added C functionality ............................................. 297
cwchar.h (library header file) ..................................... 296
C_INCLUDE (environment variable) ............................. 146–147
C-SPY
  interface to system termination ............................. 74
  STL container support ......................................... 109
C++
  See also Embedded C++ and Extended Embedded C++
  absolute location ................................................ 131
calling convention ................................................ 94
  features excluded from EC++ ................................ 105
  header files ..................................................... 294
language extensions .............................................. 110
  special function types ....................................... 36
  static function types ......................................... 110
support for ................................................................ 5
C++ terminology ..................................................... xxxiii
C++-style comments ................................................ 225
C99 standard, added functionality from .................... 296

D
-D (compiler option) .................................................. 162
d (archive option) ..................................................... 328
data
  alignment of ....................................................... 209
different ways of storing ........................................ 25
  located, declaring extern ..................................... 130
placing ................................................................. 129, 185, 321
  at absolute location ........................................... 130
representation of ................................................... 209
storage ................................................................... 25
data block (call frame information) ......................... 102
data pointers .......................................................... 215
data types ............................................................... 210
floating point .......................................................... 213
in C++ ................................................................. 219
  integers ............................................................... 210
  data alignment (pragma directive) ......................... 247
-_DATE_ (predefined symbol) .................................... 285
date ............................................................. 146–147
DC32 (assembler directive) ....................................... 92
--debug (compiler option) ........................................ 163
debug information, including in object file ............... 163, 183
  .debug (ELF section) ............................................. 322
debugger, low-level interface ................................... 84
  DebugMon_Handler (exception function) ................... 36
declarations
  empty ..................................................................... 230
  in for loops ........................................................ 225
  Kernighan & Ritchie ............................................ 139
  of functions ....................................................... 96
declarations and statements, mixing ......................... 225
declarators, implementation-defined behavior ............ 358
define block (linker directive) .................................. 305
define overlay (linker directive) .............................. 306
define symbol (linker directive) ............................... 316
--define_symbol (linker option) ............................. 193
define_type_info (pragma directive) ......................... 359
--delete (archive option) ...................................... 328
delete (keyword) .................................................... 27
denormalized numbers. See subnormal numbers
diagnostic messages ................................................ 150
classifying as compilation errors ............................ 164
classifying as compilation remarks ......................... 164
classifying as compiler warnings ......................... 165
classifying as errors ____________________________ 175, 200
classifying as linker warnings ............................... 195
classifying as linking errors ................................. 194
classifying as linking remarks ............................... 194
  disabling compiler warnings .............................. 180
  disabling linker warnings ................................. 202
disabling wrapping of in compiler ......................... 180

Index
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry label, program</td>
<td>71</td>
</tr>
<tr>
<td>enumerations, implementation-defined behavior</td>
<td>357</td>
</tr>
<tr>
<td>error messages</td>
<td>151</td>
</tr>
<tr>
<td>data representation</td>
<td>211</td>
</tr>
<tr>
<td>forward declarations of</td>
<td>229</td>
</tr>
<tr>
<td>__enum_is_int (compiler option)</td>
<td>169</td>
</tr>
<tr>
<td>environment</td>
<td></td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>354</td>
</tr>
<tr>
<td>runtime (DLIB)</td>
<td>59</td>
</tr>
<tr>
<td>environment variables</td>
<td></td>
</tr>
<tr>
<td>C_INCLUDE</td>
<td>146-147</td>
</tr>
<tr>
<td>ILINKARM_CMD_LINE</td>
<td>147</td>
</tr>
<tr>
<td>QCCARM</td>
<td>146</td>
</tr>
<tr>
<td>EQU (assembler directive)</td>
<td>183</td>
</tr>
<tr>
<td>ERANGE</td>
<td>360</td>
</tr>
<tr>
<td>errno.h (library header file)</td>
<td>293</td>
</tr>
<tr>
<td>error return codes</td>
<td>149</td>
</tr>
<tr>
<td>__error_limit (compiler option)</td>
<td>169</td>
</tr>
<tr>
<td>__error_limit (linker option)</td>
<td>196</td>
</tr>
<tr>
<td>exception handling, missing from Embedded C++</td>
<td>105</td>
</tr>
<tr>
<td>exception stacks</td>
<td>114</td>
</tr>
<tr>
<td>exception (library header file)</td>
<td>294</td>
</tr>
<tr>
<td>_Exit (library function)</td>
<td>73</td>
</tr>
<tr>
<td>exit (library function)</td>
<td>73</td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>362</td>
</tr>
<tr>
<td>__exit (library function)</td>
<td>73</td>
</tr>
<tr>
<td>_exit (library function)</td>
<td>73</td>
</tr>
<tr>
<td>export keyword, missing from Extended EC++</td>
<td>108</td>
</tr>
<tr>
<td>export (linker directive)</td>
<td>317</td>
</tr>
<tr>
<td>--export_builtin_config (linker option)</td>
<td>196</td>
</tr>
<tr>
<td>expressions (in linker configuration file)</td>
<td>317</td>
</tr>
<tr>
<td>extended command line file for compiler</td>
<td>169</td>
</tr>
<tr>
<td>for linker</td>
<td>197</td>
</tr>
<tr>
<td>passing options</td>
<td>146</td>
</tr>
<tr>
<td>Extended Embedded C++</td>
<td>106</td>
</tr>
<tr>
<td>enabling</td>
<td>167</td>
</tr>
<tr>
<td>standard template library (STL)</td>
<td>295</td>
</tr>
<tr>
<td>extended keywords</td>
<td>233-325</td>
</tr>
<tr>
<td>enabling (-e)</td>
<td>167</td>
</tr>
<tr>
<td>overview</td>
<td>6</td>
</tr>
<tr>
<td>summary</td>
<td>236</td>
</tr>
<tr>
<td>syntax</td>
<td></td>
</tr>
<tr>
<td>object attributes</td>
<td>235</td>
</tr>
<tr>
<td>type attributes on data objects</td>
<td>234</td>
</tr>
<tr>
<td>type attributes on data pointers</td>
<td>234</td>
</tr>
<tr>
<td>type attributes on functions</td>
<td>234</td>
</tr>
<tr>
<td>extended-selectors (in linker configuration file)</td>
<td>315</td>
</tr>
<tr>
<td>external &quot;C&quot; linkage</td>
<td>108</td>
</tr>
<tr>
<td>__extract (archive option)</td>
<td>328</td>
</tr>
</tbody>
</table>

**F**

- `f` (compiler option)                                             | 169  |
- `f` (archive option)                                              | 327  |
- `f` (objcmanip option)                                            | 342  |
- `f` (linker option)                                               | 197  |
- fast interrupts                                                   | 32  |
- fatal error messages                                              | 151  |
- fgetpos (library function), implementation-defined behavior       | 362  |
- __FILE__ (predefined symbol)                                      | 285  |
file dependencies, tracking ............................................ 163
file paths, specifying for #include files ......................... 171
filename
  extension for device description files ...................... 6
  extension for header files ................................. 6
  of object executable image .............................. 203
  search procedure for ...................................... 147
  specifying as parameter .................................. 154
  --fill (elftool option) ..................................... 336
  _fiq (extended keyword) .................................... 238
FIQ_Handler (exception function) ............................. 32
float (data type) ................................................. 213
floating-point constants
  hexadecimal notation ....................................... 227
  hints ......................................................... 126
floating-point expressions,
  using in preprocessor extensions ........................ 173
floating-point format .......................................... 213
  hints ......................................................... 126
  implementation-defined behavior .......................... 356
  special cases .............................................. 214
  32-bits .................................................... 213
  64-bits .................................................... 214
floating-point unit ............................................. 170
float_t, C99 extension .......................................... 297
float.h (library function),
  implementation-defined behavior ........................ 361
  for loops, declarations in .................................. 225
  --force_output (linker option) ......................... 197
formats
  floating-point values ..................................... 213
  standard IEEE (floating point) ........................... 213
  fpclassify, C99 extension ................................ 297
  --fpu (compiler option) .................................... 170
FP_INFINITE, C99 extension .................................. 297
FP_NAN, C99 extension ......................................... 297
FP_NORMAL, C99 extension .................................... 297
FP_SUBNORMAL, C99 extension ................................. 297
FP_ZERO, C99 extension ....................................... 297
fragmentation, of heap memory .................................. 27
free (library function), See also heap ......................... 27
fstream (library header file) ................................ 294
fstream.h (library header file) ............................... 294
ftell (library function), implementation-defined behavior .. 362
Full DLIB (library configuration) ........................... 61
  __func__ (predefined symbol) .................. 231, 285
  __FUNCTION__ (predefined symbol) .......... 231, 285
function calls
  calling convention ........................................... 95
  stack image after .......................................... 99
  function declarations, Kernighan & Ritchie .......... 139
  function execution, in RAM .................................. 30
  function inlining (compiler transformation) .......... 135
  disabling --no_inline ..................................... 176
  function pointers ......................................... 215
  function prototypes ....................................... 138
  enforcing .................................................. 184
  function (pragma directive), ......................... 359
  functional (STL header file) ............................. 295
  functions .................................................. 29
  C++ and special function types ......................... 36
  declaring ................................................... 96, 138
  inlining ..................................................... 135, 137, 225, 250
  interrupt .................................................... 31
  intrinsic .................................................... 89, 138
  parameters .................................................. 97
  placing in memory ......................................... 129, 131, 185
  recursive
    avoiding ....................................................... 138
    storing data on stack ..................................... 26–27
  reentrancy (DLIB) ........................................... 292
  related extensions ......................................... 29
  return values from ......................................... 99
  special function types ..................................... 30
getenv (library function), configuring support ............. 81

G

IAR C/C++ Development Guide
372  Compiling and linking for ARM
| getzone (library function), configuring support for | 83 |
| getzone.c | 83 |
| __get_BASEPRI (intrinsic function) | 264 |
| __get_CONTROL (intrinsic function) | 264 |
| __get_CPSR (intrinsic function) | 265 |
| __get_FAULTMASK (intrinsic function) | 265 |
| __get_INTERRUPT_state (intrinsic function) | 265 |
| __get_LR (intrinsic function) | 265 |
| __get_PRIMASK (intrinsic function) | 266 |
| __get_SB (intrinsic function) | 266 |
| __get_SP (intrinsic function) | 266 |
| GRP_COMDAT, group type | 345 |
| guidelines, reading | xxix |
| Hashing | 295 |
| hardware support in compiler | 60 |

**H**

Harbison, Samuel P. | xxxii |
HardFault_Handler (exception function) | 36 |
hardware support in compiler | 60 |
hash_map (STL header file) | 295 |
hash_set (STL header file) | 295 |
hdrtstop (pragma directive) | 359 |
header files | |
C | 293 |
C++ | 294 |
EC++ | 294 |
library | 291 |
special function registers | 140 |
STL | 295 |
DLib_Config_Full.h | 69 |
DLib_Config_Normal.h | 69 |
DLib_Defaults.h | 69 |
intrinsics.h | 259 |
stbool.h | 211, 293 |
stddf.h | 212 |
--header_context (compiler option) | 171 |
heap | |
dynamic memory | 27 |
storing data | 25 |

heap size
and standard I/O | 115 |
changing default | 52 |
HEAP (section) | 115, 323 |
hide (isymexport directive) | 348 |
hints
for good code generation | 137 |
using efficient data types | 125 |
HUGE_VALF, C99 extension | 297 |
HUGE_VALL, C99 extension | 297 |

-I (compiler option) | 171 |
IAR Command Line Build Utility | 69 |
IAR Systems Technical Support | 152 |
 iarbuild.exe (utility) | 69 |
 iarchive | 325 |
 commands summary | 326 |
 options summary | 327 |
 __IAR_SYSTEMS_ICC__ (predefined symbol) | 285 |
 .iar.debug (ELF section) | 322 |
 .iar.dynexit (section) | 323 |
 IA64 ABI | 121 |
 __ICCARM__ (predefined symbol) | 285 |
 icons, in this guide | xxxiii |
IDE
building a library from | 69 |
overview of build tools | 3 |
identifiers, implementation-defined behavior | 354 |
IEEE format, floating-point values | 213 |
telfdump | 338 |
options summary | 339 |
telftool | 332 |
options summary | 333 |
if (linker directive) | 319 |
--iHex (ielftool option) | 336 |
ILINK
See linker
### IAR C/C++ Development Guide

**Compiling and linking for ARM**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILINK options. See linker options</td>
<td>153</td>
</tr>
<tr>
<td>Intel hex</td>
<td>113</td>
</tr>
<tr>
<td>Intel IA64 ABI</td>
<td>121</td>
</tr>
<tr>
<td>internal error</td>
<td>152</td>
</tr>
<tr>
<td>interrupt functions</td>
<td>31</td>
</tr>
<tr>
<td>fast interrupts</td>
<td>32</td>
</tr>
<tr>
<td>in ARM Cortex</td>
<td>36</td>
</tr>
<tr>
<td>nested interrupts</td>
<td>33</td>
</tr>
<tr>
<td>operations</td>
<td>35</td>
</tr>
<tr>
<td>software interrupts</td>
<td>34</td>
</tr>
<tr>
<td>interrupt state, restoring</td>
<td>273</td>
</tr>
<tr>
<td>interrupt vector table</td>
<td>35</td>
</tr>
<tr>
<td>.intvec section</td>
<td>324</td>
</tr>
<tr>
<td>interrupts</td>
<td></td>
</tr>
<tr>
<td>processor state</td>
<td>26</td>
</tr>
<tr>
<td>using with EC++ destructors</td>
<td>110</td>
</tr>
<tr>
<td>__interwork (extended keyword)</td>
<td>238</td>
</tr>
<tr>
<td>--interwork (compiler option)</td>
<td>171</td>
</tr>
<tr>
<td>interworking code</td>
<td>21</td>
</tr>
<tr>
<td>intptr_t (integer type)</td>
<td>216</td>
</tr>
<tr>
<td>__intrinsic (extended keyword)</td>
<td>238</td>
</tr>
<tr>
<td>intrinsic functions</td>
<td>138</td>
</tr>
<tr>
<td>overview</td>
<td>89</td>
</tr>
<tr>
<td>summary</td>
<td>259</td>
</tr>
<tr>
<td>intrinsics.h (header file)</td>
<td>259</td>
</tr>
<tr>
<td>introduction</td>
<td></td>
</tr>
<tr>
<td>linker configuration file</td>
<td>299</td>
</tr>
<tr>
<td>linking</td>
<td>39</td>
</tr>
<tr>
<td>inttypes.h (library header file)</td>
<td>293</td>
</tr>
<tr>
<td>inttypes.h, added C functionality</td>
<td>297</td>
</tr>
<tr>
<td>.intvec (section)</td>
<td>324</td>
</tr>
<tr>
<td>invocation syntax</td>
<td>145</td>
</tr>
<tr>
<td>iobjmanip</td>
<td>341</td>
</tr>
<tr>
<td>options summary</td>
<td>342</td>
</tr>
<tr>
<td>iomanip (library header file)</td>
<td>294</td>
</tr>
<tr>
<td>iomanip.h (library header file)</td>
<td>294</td>
</tr>
<tr>
<td>ios (library header file)</td>
<td>294</td>
</tr>
<tr>
<td>iosfwd (library header file)</td>
<td>294</td>
</tr>
<tr>
<td>iostream (library header file)</td>
<td>294</td>
</tr>
<tr>
<td>iostream.h (library header file)</td>
<td>294</td>
</tr>
</tbody>
</table>
_irq (extended keyword) .......................... 238
IRQ_Handler (exception function) .................. 32
IRQ_STACK (section) .............................. 324
_isb (intrinsic function) .......................... 266
isfinite, C99 extension .......................... 297
isgreater, C99 extension .......................... 297
isinf, C99 extension .............................. 297
isessequal, C99 extension .......................... 297
islessgreater, C99 extension ...................... 297
isless, C99 extension .............................. 297
isnan, C99 extension .............................. 297
isnormal, C99 extension .......................... 297
isunordered, C99 extension .......................... 297
I/O module, overriding in runtime library ........ 67
ISO/ANSI C
  compiler extensions ............................ 221
  C++ features excluded from EC++ ................. 105
    specifying strict usage ......................... 186
isof646.h (library header file) .................. 293
istream (library header file) .................... 294
isunordered, C99 extension .......................... 297
iswblank, C99 extension .......................... 297
isymexport ....................................... 346
  options summary .................................. 347
italic style, in this guide ........................ xxxiii
I/O debugging, support for .......................... 84
I/O module, overriding in runtime library ........ 67

K

--keep (linker option) .......................... 198
keep (linker directive) ........................... 310
keep_definition (pragma directive) .............. 360
Kernighan & Ritchie function declarations .......... 139
  disallowing ...................................... 184
Kernighan, Brian W. ................................ xxxii
keywords ......................................... 233
  extended, overview of .......................... 6

L

-1 (compiler option) ................................ 172
  for creating skeleton code ....................... 93
labels ............................................. 230
  assembler, making public ....................... 183
  __program_start .................................. 71
Labrosse, Jean J. ................................ xxxii
Lajoie, Josée ................................ xxxii
language extensions
  descriptions ....................................... 221
  Embedded C++ .................................... 105
  enabling ......................................... 251
  enabling (-e) ..................................... 167
language overview ................................... 5
language (pragma directive) ........................ 251
  _LDC (intrinsic function) ....................... 266
  _LDCL (intrinsic function) ..................... 266
  _LDCL_noidx (intrinsic function) .............. 267
  _LDC_noidx (intrinsic function) ............... 267
  _LDC2 (intrinsic function) ..................... 266
  _LDC2L (intrinsic function) .................... 266
  _LDC2L_noidx (intrinsic function) ............ 267
  _LDC2_noidx (intrinsic function) .............. 267
  _LDREX (intrinsic function) .................... 267
  --legacy (compiler option) ...................... 173
Lempel-Ziv-Welch algorithm, for packing initializers .... 308
  for creating skeleton code ...................... 93
  standard template library ....................... 295
library configuration files
  DLIB ............................................. 61
  DLlib_Defaults.h ................................ 69
  for Normal and Full ................................ 69
  modifying ........................................ 69
  specifying ........................................ 166
library documentation................................ 291
library features, missing from Embedded C++ ........ 106

375
malloc (library function)

See also heap ........................................ 27
implementation-defined behavior ............. 362
--mangled_names_in_messages (linker option) ... 199
Mann, Bernhard ..................................... xxxii
map file, producing ............................... 200
map (STL header file) .............................. 295
MATH_ERREXCEPT, C99 extension ............... 297
math_errhandling, C99 extension ................. 297
math.h (library header file) ....................... 293
math.h, added C functionality ................... 297
__MCR (intrinsic function) ......................... 267
member functions, pointers to ................. 109
MemManage_Handler (exception function) .... 36
memory
allocating in C++ .................................. 27
dynamic ............................................. 27
heap .................................................. 27
non-initialized ...................................... 141
RAM, saving ........................................ 138
releasing in C++ .................................... 27
stack ................................................ 26
saving ............................................... 138
used by global or static variables .......... 25
memory layout, ARM ................................ 25
memory management, type-safe ................ 105
memory map
  generating ....................................... 200
  output from linker .............................. 149
memory (pragma directive) ....................... 360
memory (STL header file) ......................... 295
message (pragma directive) ....................... 252
messages
  disabling ....................................... 186, 206
  forcing .......................................... 252
--mfc (compiler option) ......................... 173
--migration_preprocessor_extensions (compiler option) ... 173
--misrac_verbose (compiler option) ........... 157
--misrac1998 (linker option) .................... 157
--misrac1998 (compiler option) ................. 157
--misrac2004 (linker option) .................... 157
--misrac2004 (compiler option) ................. 157
module consistency ................................ 85
rmmodel ........................................... 255
modules, introduction ........................... 40
module_name (pragma directive) ................. 360
Motorola S-records ................................ 113
__MRC (intrinsic function) ....................... 268
multibyte character support ..................... 168
multiple inheritance
  in Extended EC++ ................................ 106
  missing from Embedded C++ ..................... 105
  missing from STL ................................ 106
multi-file compilation ........................... 133
mutable attribute, in Extended EC++ ......... 106, 109

N
names block (call frame information) ........... 101
namespace support
  in Extended EC++ ................................ 106, 109
  missing from Embedded C++ ..................... 106
naming conventions ................................ xxxiv
NAN, C99 extension ................................ 297
NDEBUG (preprocessor symbol) .................... 287
__nested (extended keyword) ...................... 239
nested interrupts ................................ 33
new (keyword) ..................................... 27
new (library header file) ......................... 294
new.h (library header file) ....................... 294
NMI_Handler (exception function) ............... 294
NMI_Handler (exception function) ............... 36
non-initialized variables, hints for .......... 141
non-scalar parameters, avoiding ............... 138
NOP (assembler instruction) ...................... 268
__noretum (extended keyword) .................... 240
Normal DLIB (library configuration) .......... 61

377
Not a number (NaN) .............................................. 214
--no_clustering (compiler option) ................................ 174
--no_code_motion (compiler option) ................................ 174
--no_const_align (compiler option) ................................ 175
--no_cse (compiler option) ......................................... 175
--no_guard_calls (compiler option) ................................. 176
__no_init (extended keyword) ....................................... 141, 239
--no_inline (compiler option) ....................................... 176
--no_library_search (linker option) ................................ 201
--no_locals (linker option) ......................................... 201
__no_operation (intrinsic function) ................................. 268
--no_path_in_file Macros (compiler option) ......................... 176
no_pch (pragma directive) ........................................... 360
--no_remove (linker option) ......................................... 201
--no_rwlock_dynamic_init (compiler option) ......................... 177
--no_scheduling (compiler option) ................................... 177
--no_typeof_in_diagnostics (compiler option) ...................... 178
--no_unaligned_access (compiler option) ........................... 179
--no_unroll (compiler option) ....................................... 179
--no_veneer (linker option) ......................................... 202
--no_warnings (compiler option) .................................... 180
--no_warnings (linker option) ....................................... 202
--no_wrap_diagnostics (compiler option) ............................ 180
--no_wrap_diagnostics (linker option) ................................ 202
NULL (macro), implementation-defined behavior ..................... 360
numbers (in linker configuration file) ............................. 318
numeric (STL header file) ............................................ 295

-O (compiler option) .............................................. 180
-o (compiler option) ............................................... 181
-o (iarchive option) ................................................. 329
-o (ielfdump option) ................................................. 340
-o (linker option) .................................................. 202
object attributes ..................................................... 235
object filename
    specifying ....................................................... 340
    specifying in compiler ....................................... 181

specifying in linker .............................................. 202
object module, ose .................................................. 203
object_attribute (pragma directive) ................................ 141, 252
once (pragma directive) ............................................. 360
--only_stdout (linker option) ...................................... 181
--only_stdout (compiler option) .................................... 203
__open (library function) ............................................ 79
operators
    See also @ (operator)
optimization
    clustering, disabling ............................................ 174
    code motion, disabling .......................................... 174
    common sub-expression elimination, disabling ............... 175
    configuration ...................................................... 21
    disabling .......................................................... 134
    function inlining, disabling (--no_inline) .................. 176
    hints ............................................................... 137
    loop unrolling, disabling ....................................... 179
    scheduling, disabling ............................................ 177
    specifying (-O) ................................................... 180
    techniques ........................................................ 134
    type-based alias analysis, disabling (--tbaa) ............... 178
    using inline assembler code .................................... 92
    using pragma directive ......................................... 252
optimization levels ................................................. 133
optimize (pragma directive) ........................................ 252
option parameters ................................................... 153
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
Oram, Andy ............................................................ xxxii
--ose_load_module (linker option) ................................ 203
output
    from preprocessor .............................................. 182
_printf_args (pragma directive) ................................. 254
processor configuration ........................................... 20
processor operations
   accessing ....................................................... 89
   low-level ..................................................... 221, 259
program entry label ............................................. 71
programming hints .............................................. 137
__program_start (label) ......................................... 71
projects
   basic settings for ........................................... 19
   setting up for a library ................................... 69
prototypes, enforcing ........................................... 184
ptrdiff_t (integer type) ......................................... 216
PUBLIC (assembler directive) ................................. 183
publication date, of this guide ................................ . ii
--public_equ (compiler option) ................................. 183
public_eq (pragma directive), absent from DLIB .......... 360
putenv (library function), absent from DLIB ............. 81

R

- (compiler option) ............................................. 183
-r (archive option) ............................................. 329
raise (library function), configuring support for .......... 82
raise.c .......................................................... 82
RAM
   example of declaring region .................................. 43
   execution ..................................................... 30
   initializers copied from ROM ................................ 18
   running code from ........................................... 54
   saving memory ............................................... 138
   _ramfunc (extended keyword) ................................ 30, 241
   range errors .................................................. 56
   --raw (elfdump option) ..................................... 341
   _read (library function) ..................................... 79
   customizing .................................................. 75
   read formatter, selecting ................................... 67
   reading guidelines ........................................... xxxix
   reading, recommended ....................................... xxxii
   realloc (library function)
      implementation-defined behavior ......................... 362
      See also heap .............................................. 27
   recursive functions
      avoiding ..................................................... 138
      storing data on stack ..................................... 26–27
      --redirect (linker option) ................................. 205
   reentrancy (DLIB) .............................................. 292
   reference information, typographic convention .......... xxxiii
   region .......................................................... 43
   region expression, in linker configuration file .......... 303
   region literal, in linker configuration file .............. 301
   register parameters .......................................... 97–98
   registered trademarks ....................................... ii
   registers
      assigning to parameters .................................... 98
      callee-save, stored on stack ............................... 26
      implementation-defined behavior .......................... 357
      in assembler-level routines ............................... 95

Q

__QADD (intrinsic function) ................................. 269
__QADD8 (intrinsic function) .................................. 269
__QADD16 (intrinsic function) ................................ 269
__QASX (intrinsic function) .................................... 269
QCCARM (environment variable) ............................ 146
__QDADD (intrinsic function) .................................. 269
__QDOUBLE (intrinsic function) ................................ 270
__QDSUB (intrinsic function) .................................. 270
__QFlag (intrinsic function) ................................... 270
__QSAX (intrinsic function) .................................... 271
__QSUB (intrinsic function) .................................... 270
__QSUB8 (intrinsic function) ................................... 271
__QSUB16 (intrinsic function) .................................. 271
qualifiers
   const and volatile ........................................... 218
   implementation-defined behavior .......................... 358
   queue (STL header file) ..................................... 295
preserved ........................................... 97
scratch ........................................... 97
reinterpret_cast (cast operator) .................. 106
.rel (ELF section) .................................. 322
.rela (ELF section) ................................. 322
relocation errors, resolving ....................... 57
remark (diagnostic message) ....................... 151
classifying for compiler .......................... 164
classifying for linker .............................. 194
enabling in compiler .............................. 183
enabling in linker ................................. 205
-- remarks (compiler option) ...................... 183
-- remarks (linker option) ....................... 205
remove (library function) ......................... 79
implementation-defined behavior .................. 361
--remove_section (iobjmanip option) .......... 343
rename (isymexport directive) ................... 348
rename (library function) ......................... 79
implementation-defined behavior .................. 361
--rename_section (iobjmanip option) ........... 343
--rename_symbol (iobjmanip option) .......... 343
--replace (archive option) ....................... 329
__ReportAssert (library function) ............... 83
required (pragma directive) ..................... 254
-- require_prototypes (compiler option) ....... 184
return values, from functions ................... 99
__REV (intrinsic function) ....................... 271
__REVSH (intrinsic function) ..................... 271
Richie, Dennis M. .................................. xxxii
ROM to RAM, copying ................................ 54
__root (extended keyword) ....................... 241
__ROPI__ (predefined symbol) .................... 286
--ropi (compiler option) ......................... 184
routines, time-critical .......................... 89, 221, 259
tmodel (assembler directive) ...................... 87
tmodel (pragma directive) ......................... 255
rtti support, missing from STL ................... 106
runtime environment ................................ 59
setting libraries .................................... 23
choosing ........................................... 23
introduction ........................................ 291
DLIB ............................................... 62
choosing ........................................... 62
customizing without rebuilding ................... 64
naming convention .................................. 63
overriding modules in ............................. 67
runtime model attributes .......................... 85
runtime model definitions .......................... 255
runtime type information, missing from Embedded C++ ........... 106
--rwpi (compiler option) ......................... 185
S
-S (archive option) ................................ 329
-s (ielfdump option) ............................... 340
__SADD8 (intrinsic function) ..................... 272
__SADD16 (intrinsic function) .................... 272
__SASX (intrinsic function) ...................... 272
__shrel (extended keyword) ....................... 236
scanf (library function) ......................... 23
configuration symbols ............................ 77
implementation-defined behavior .................. 362
scanf (library function), choosing formatter .... 66
__scanf_args (pragma directive) ................. 256
scheduling (compiler transformation) ............ 137
disabling .......................................... 177
scratch registers ................................... 97
--section (ielfdump option) ..................... 340
section names ...................................... 256
declaring .......................................... 256
specifying .......................................... 185
section (pragma directive) ....................... 256
sections ........................................... 321
summary ........................................... 321
allocation of ....................................... 42
introduction ........................................ 40
std namespace, missing from EC++
and Extended EC++ .............................. 109
stdarg.h (library header file) .................. 293
stdbool.h (library header file) ................. 211, 293
added C functionality ........................... 297
__STDC__ (predefined symbol) ................... 286
__STDC_VERSION__ (predefined symbol) ........ 286
stddef.h (library header file) ................ 212, 293
stderr ................................. 79, 181, 203
stdexcept (library header file) ............... 294
stdin .................................. 79
implementation-defined behavior ................ 361
stdint.h (library header file) ................ 293, 296
stdlib.h, added C functionality ............... 297
stdio.h (library header file) .................. 293
stdio.h, additional C functionality ........... 297
stdlib.h (library header file) ................ 293
stdlib.h, additional C functionality ........... 298
stdout .................................. 79, 181, 203
implementation-defined behavior ................ 361
Stein, Guy L. .................................. xxxii
steering file, input to ismexport ............... 347
STL ........................................ 109
streambuf (library header file) ............... 294
streams, supported in Embedded C++ ............ 106
stream (library header file) .................. 294
stream.h (library header file) ................ 293
 stiff (ielftool option) ......................... 338
--strip (linker option) ......................... 206
Stevens, Steve ................................ 383

Index

static_cast (cast operator) ....................... 106
static variables .................................. 25
taking the address of .............................. 137
static_cast (cast operator) ....................... 106
__STC (intrinsic function) ....................... 276
__STCL (intrinsic function) ..................... 276
__STC_noidx (intrinsic function) ................ 277
__STC (intrinsic function) ....................... 276
__STCL (intrinsic function) ..................... 276
__STC2L (intrinsic function) .................... 276
__STC2L_noidx (intrinsic function) .............. 277
__STC2L (intrinsic function) .................... 276
__STC2L_noidx (intrinsic function) .............. 277
__STC (intrinsic function) ....................... 276
__STCL (intrinsic function) ..................... 276
__STC2L (intrinsic function) .................... 276
__STC2L_noidx (intrinsic function) .............. 277
__STC2L_noidx (intrinsic function) .............. 277

stack ........................................ 113
stack parameters ................................ 97, 99
stack pointer .................................... 26
stack (STL header file) ......................... 295
standard error 
  redirecting in compiler ......................... 181
  redirecting in linker .......................... 203
standard input .................................. 75
standard output .................................. 75
specifying in compiler ........................... 181
specifying in linker .............................. 203
standard template library (STL) 
in Extended EC++ ............................... 106, 109, 295
missing from Embedded C++ ...................... 106
startup system. See system startup
statements, implementation-defined behavior ...... 358
static clustering (compiler transformation) ..... 136
static variables .................................. 25

Index
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>strtod, in stdlib.h</td>
<td>298</td>
</tr>
<tr>
<td>strtol, C99 extension</td>
<td>298</td>
</tr>
<tr>
<td>strtold, C99 extension</td>
<td>298</td>
</tr>
<tr>
<td>strtoul, C99 extension</td>
<td>298</td>
</tr>
<tr>
<td>system startup</td>
<td>322</td>
</tr>
<tr>
<td>system syntax</td>
<td>322</td>
</tr>
<tr>
<td>.symtab (ELF section)</td>
<td>322</td>
</tr>
<tr>
<td>--symbols (iarchive option)</td>
<td>330</td>
</tr>
<tr>
<td>symbol names, using in preprocessor extensions</td>
<td>174</td>
</tr>
<tr>
<td>structure types</td>
<td>216–217</td>
</tr>
<tr>
<td>alignment</td>
<td>216</td>
</tr>
<tr>
<td>layout of</td>
<td>217</td>
</tr>
<tr>
<td>packed</td>
<td>217</td>
</tr>
<tr>
<td>structures</td>
<td>216</td>
</tr>
<tr>
<td>aligning</td>
<td>254</td>
</tr>
<tr>
<td>anonymous</td>
<td>127, 223</td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>357</td>
</tr>
<tr>
<td>incomplete arrays as last element</td>
<td>226</td>
</tr>
<tr>
<td>packing and unpacking</td>
<td>127</td>
</tr>
<tr>
<td>subnormal numbers</td>
<td>213–214</td>
</tr>
<tr>
<td>support, technical</td>
<td>152</td>
</tr>
<tr>
<td>SVC #immed, for software interrupts</td>
<td>34</td>
</tr>
<tr>
<td>SVC_Handler (exception function)</td>
<td>36</td>
</tr>
<tr>
<td>__swi (extended keyword)</td>
<td>242</td>
</tr>
<tr>
<td>SWI_Handler (exception function)</td>
<td>36</td>
</tr>
<tr>
<td>swi_number (pragma directive)</td>
<td>32</td>
</tr>
<tr>
<td>__SWP (intrinsic function)</td>
<td>278</td>
</tr>
<tr>
<td>__SWPB (intrinsic function)</td>
<td>278</td>
</tr>
<tr>
<td>symbol names, using in preprocessor extensions</td>
<td>174</td>
</tr>
<tr>
<td>symbols</td>
<td>226</td>
</tr>
<tr>
<td>anonymous, creating</td>
<td>205</td>
</tr>
<tr>
<td>directing from one to another</td>
<td>254</td>
</tr>
<tr>
<td>including in output</td>
<td>254</td>
</tr>
<tr>
<td>overview of predefined</td>
<td>7</td>
</tr>
<tr>
<td>preprocessor, defining</td>
<td>162, 193</td>
</tr>
<tr>
<td>--symbols (iarchive option)</td>
<td>330</td>
</tr>
<tr>
<td>.symtab (ELF section)</td>
<td>322</td>
</tr>
<tr>
<td>syntax</td>
<td></td>
</tr>
<tr>
<td>command line options</td>
<td>153</td>
</tr>
<tr>
<td>extended keywords</td>
<td>234–235</td>
</tr>
<tr>
<td>invoking compiler and linker</td>
<td>145</td>
</tr>
<tr>
<td>system startup</td>
<td>74</td>
</tr>
<tr>
<td>customizing</td>
<td>74</td>
</tr>
<tr>
<td>DLIB</td>
<td>70</td>
</tr>
<tr>
<td>initialization phase</td>
<td>15</td>
</tr>
<tr>
<td>system termination</td>
<td></td>
</tr>
<tr>
<td>C-SPY interface to</td>
<td>74</td>
</tr>
<tr>
<td>DLIB</td>
<td>73</td>
</tr>
<tr>
<td>system (library function)</td>
<td></td>
</tr>
<tr>
<td>configuring support for</td>
<td>81</td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>363</td>
</tr>
<tr>
<td>system_include (pragma directive)</td>
<td>360</td>
</tr>
<tr>
<td>SysTick_Handler (exception function)</td>
<td>36</td>
</tr>
<tr>
<td>Terminal I/O window</td>
<td></td>
</tr>
<tr>
<td>making available</td>
<td>85</td>
</tr>
<tr>
<td>terminal output, speeding up</td>
<td>85</td>
</tr>
<tr>
<td>termination of system</td>
<td>xxxiii</td>
</tr>
<tr>
<td>See system termination terminology</td>
<td></td>
</tr>
<tr>
<td>32-bits (floating-point format)</td>
<td>213</td>
</tr>
<tr>
<td>this (pointer)</td>
<td>94</td>
</tr>
<tr>
<td>__thumb (extended keyword)</td>
<td>243</td>
</tr>
<tr>
<td>-thumb (compiler option)</td>
<td>187</td>
</tr>
<tr>
<td>Thumb, CPU mode</td>
<td>20</td>
</tr>
<tr>
<td><strong>TID</strong> (predefined symbol)</td>
<td>286</td>
</tr>
<tr>
<td><strong>TIME</strong> (predefined symbol)</td>
<td>286</td>
</tr>
<tr>
<td>time zone (library function)</td>
<td></td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>363</td>
</tr>
<tr>
<td>time (library function), configuring support for</td>
<td>83</td>
</tr>
<tr>
<td>time-critical routines</td>
<td>89, 221, 259</td>
</tr>
<tr>
<td>time.e</td>
<td>83</td>
</tr>
<tr>
<td>time.h (library header file)</td>
<td>293</td>
</tr>
<tr>
<td>tips, programming</td>
<td>137</td>
</tr>
<tr>
<td>--toc (iarchive option)</td>
<td>330</td>
</tr>
</tbody>
</table>
tools icon, in this guide ........................................ xxxiii
trademarks ...................................................... ii
transformations, compiler .................................. 132
translation, implementation-defined behavior .......... 353
type attributes ............................................. 233
specifying ..................................................... 257
type qualifiers
const and volatile ........................................... 218
implementation-defined behavior ....................... 358
typedefsncluding from diagnostics ....................... 178
repeated ....................................................... 229
using in preprocessor extensions ....................... 174
type_attribute (pragma directive) ...................... 257
type-based alias analysis (compiler transformation) .. 136
disabling ..................................................... 178
type-safe memory management ......................... 105
typographic conventions ................................ xxxiii

U
__UADD8 (intrinsic function) ......................... 278
__UADD16 (intrinsic function) ..................... 278
__UASX (intrinsic function) ....................... 278
__UHADD8 (intrinsic function) ................. 279
__UHADD16 (intrinsic function) ........... 279
__UHASX (intrinsic function) .............. 279
__UHSAAX (intrinsic function) .......... 279
__UHSAX (intrinsic function) ............ 279
__UHSUB8 (intrinsic function) ...... 279
__UHSUB16 (intrinsic function) .... 280
uintptr_t (integer type) ....................... 216
Undefined_Handler (exception function) ........... 32
underflow range errors,
implementation-defined behavior ................. 360
unions
anonymous .............................................. 127, 223
implementation-defined behavior .............. 357
unsigned char (data type) .................... 210–211
changing to signed char ......................... 160
unsigned int (data type) ...................... 211
unsigned long long (data type) ............ 211
unsigned long (data type) ..................... 211
unsigned short (data type) .................... 210
__UQADD8 (intrinsic function) .......... 280
__UQADD16 (intrinsic function) ........ 280
__UQASX (intrinsic function) .......... 280
__UQSAAX (intrinsic function) ....... 280
__UQSUB8 (intrinsic function) ...... 281
__UQSUB16 (intrinsic function) .... 281
UsageFault_Handler (exception function) .......... 36
__USAX (intrinsic function) ............ 281
-use_unix_directory_separators (compiler option) .... 187
__USUB8 (intrinsic function) ...... 281
__USUB16 (intrinsic function) .... 282
utility (STL header file) .................... 295

V
-V (iarchive option) ........................................ 330
variables
auto ....................................................... 26
defined inside a function ......................... 26
global, placement in memory .................. 25
hints for choosing .................................. 137
local. See auto variables
non-initialized ..................................... 141
placing at absolute addresses ................ 131
placing in named sections ..................... 131
static
placement in memory ................................ 25
taking the address of ................................. 137
vector floating-point unit ...................... 170
vector (pragma directive) .................... 360
vector (STL header file) ..................... 295
__vector_table, array holding vector table .......... 36
veneers ................................................... 56
__VER__ (predefined symbol) ............. 286
-verbose (iarchive option) ..................... 330
--verbose (ielftool option) ................................................. 338
version
IAR Embedded Workbench .............................................. ii
of compiler ................................................................. 286
VFP ............................................................ 170
vfscanf, C99 extension ................................................... 297
vfscanf, C99 extension ................................................... 297
void, pointers to ......................................................... 229
volatile (keyword) ......................................................... 140
volatile, declaring objects .............................................. 218
vscanf, C99 extension ..................................................... 297
vsnprintf, C99 extension ................................................. 297
vsscanf, C99 extension ..................................................... 297
vswscanf, C99 extension .................................................. 298
vwscanf, C99 extension .................................................... 298
--warnings_affect_exit_code (compiler option) .................. 149
--warnings_affect_exit_code (linker option) ...................... 206
--warnings_are_errors (compiler option) ......................... 206
--warnings_are_errors (linker option) ............................. 206
warnings icon, in this guide ........................................... xxxiv
warnings (pragma directive) ........................................... 360
_warnings_affect_exit_code (compiler option) .................. 149, 187
_warnings_affect_exit_code (linker option) ....................... 206
_warnings_are_errors (compiler option) .......................... 187
_warnings_are_errors (linker option) ............................... 206
wchar_t (data type), adding support for in C ..................... 211
wchar.h (library header file) ........................................... 293, 296
wchar.h, added C functionality ....................................... 298
w 默认 (extended keyword) .............................................. 244
weak (pragma directive) .................................................. 258
web sites, recommended ................................................. xxxii
_write (library function) ................................................ 79
_customizing .............................................................. 75
_x
-x (iarchive option) ....................................................... 328
xreportassert.c ........................................................... 83
_Z
zeros, packing algorithm for initializers ............................ 308

Symbols

_AEABI_PORTABILITY_LEVEL (preprocessor symbol) ......... 123
_AEABI_PORTABLE (preprocessor symbol) ..................... 123
_Exit (library function) ............................................... 73
_exit (library function) ................................................. 73
_exit, C99 extension ..................................................... 298
Pragma (predefined symbol) .......................................... 288
_absolute (extended keyword) ......................................... 237
_ALIGNOF_ (operator) .................................................... 223
_arm (extended keyword) ................................................ 237
_ARMVFPV1___ (predefined symbol) ............................... 284
_ARMVFPV2___ (predefined symbol) ............................... 284
_ARMVFPV3___ (predefined symbol) ............................... 284
_ARMVFPV4___ (predefined symbol) ............................... 284
_ARMVFP_D6___ (predefined symbol) .............................. 284
_ARMVFP_FP16___ (predefined symbol) ......................... 284
_ARMVFP_SP___ (predefined symbol) .............................. 284
_ARMVFP___ (predefined symbol) .................................. 284
_ARM_PROFILE_M___ (predefined symbol) ...................... 284
_ARM4M___ (predefined symbol) .................................. 284
_ARM4TM___ (predefined symbol) .................................. 284
_ARM5E___ (predefined symbol) .................................. 284
--error_limit (linker option) .......................... 196
--export_builtin_config (linker option) .......................... 196
--extract (archive option) .......................... 328
--fill (elftool option) .......................... 336
--force_output (linker option) .......................... 197
--fp (compiler option) .......................... 170
--header_context (compiler option) .......................... 171
--image_input (linker option) .......................... 197
--interwork (compiler option) .......................... 171
--keep (linker option) .......................... 198
--legacy (compiler option) .......................... 173
--log (linker option) .......................... 198
--log_file (linker option) .......................... 199
--mangled_names_in_messages (linker option) .......................... 199
--map (linker option) .......................... 200
--mfc (compiler option) .......................... 170
--migration_preprocessor_extensions (compiler option) .......................... 173
--misrac_verbose (linker option) .......................... 190
--misrac_verbose (compiler option) .......................... 173
--misrac (compiler option) .......................... 173
--misrac1998 (linker option) .......................... 190
--misrac2004 (compiler option) .......................... 175
--misrac2004 (linker option) .......................... 190
--no_clustering (compiler option) .......................... 174
--no_code_motion (compiler option) .......................... 174
--no_const_align (compiler option) .......................... 175
--no_cse (compiler option) .......................... 175
--no_fragments (compiler option) .......................... 175
--no_fragments (linker option) .......................... 200
--no_guard_calls (compiler option) .......................... 176
--no_inline (compiler option) .......................... 176
--no_library_search (linker option) .......................... 201
--no_locals (linker option) .......................... 201
--no_path_in_file_macros (compiler option) .......................... 176
--no_remove (linker option) .......................... 201
--no_rw_dynamic_init (compiler option) .......................... 177
--no_scheduling (compiler option) .......................... 177
--no_tcba (compiler option) .......................... 178
--no_typedefs_in_diagnostics (compiler option) .......................... 178
--no_unaligned_access (compiler option) .......................... 179
--no_unroll (compiler option) .......................... 179
--no_veneer (linker option) .......................... 202
--no_warnings (compiler option) .......................... 180
--no_warnings (linker option) .......................... 202
--no_wrap_diagnostics (compiler option) .......................... 180
--no_wrap_diagnostics (linker option) .......................... 202
--only_stdout (compiler option) .......................... 181
--output (compiler option) .......................... 181
--output (linker option) .......................... 203
--pi_veneer (linker option) .......................... 204
--place_holder (linker option) .......................... 204
--predef_macro (compiler option) .......................... 182
--preinclude (compiler option) .......................... 182
--preprocess (compiler option) .......................... 182
--raw (elfdump option) .......................... 341
--redirect (linker option) .......................... 205
--rename_symbol (archive manipulate option) .......................... 343
--replace (archive option) .......................... 329
--require_prototypes (compiler option) .......................... 184
--ropi (compiler option) .......................... 184
--rwpi (compiler option) .......................... 185
--section (compiler option) .......................... 185
--section (elfdump option) .......................... 340
--semihosting (linker option) .......................... 205
--separate_cluster_for_initialized_variables (compiler option) .......................... 186
--silent (compiler option) .......................... 186
--silent (archive option) .......................... 329
--silent (elftool option) .......................... 337
--simple (elftool option) .......................... 337
--srec (elftool option) .......................... 337

IAR C/C++ Development Guide
390  Compiling and linking for ARM

DARM-5