

# IAR Embedded Workbench<sup>®</sup>

IAR C/C++ Compiler

Reference Guide

for the

**CRI6C Microprocessor Family**



CRI6C-4

**IAR**  
SYSTEMS

## **COPYRIGHT NOTICE**

© 2002–2013 IAR Systems AB.

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

## **DISCLAIMER**

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

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

## **TRADEMARKS**

IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, The Code to Success, IAR KickStart Kit, I-jet, I-scope, IAR and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

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

## **EDITION NOTICE**

Fourth edition: April 2013

Part number: CR16C-4

This guide applies to version 3.x of IAR Embedded Workbench® for the CR16C microprocessor family.

Internal reference: M13, csrct2010.1, V\_110411, IJOA.

# Brief contents

Tables .....	23
Preface .....	25
<b>Part 1. Using the compiler</b> .....	<b>33</b>
Getting started .....	35
Data storage .....	43
Functions .....	55
Placing code and data .....	65
The DLIB runtime environment .....	79
The CLIB runtime environment .....	113
Assembler language interface .....	121
Using C .....	145
Using C++ .....	153
Efficient coding for embedded applications .....	169
<b>Part 2. Reference information</b> .....	<b>187</b>
External interface details .....	189
Compiler options .....	195
Data representation .....	227
Extended keywords .....	239
Pragma directives .....	251
Intrinsic functions .....	269
The preprocessor .....	277

Library functions .....	283
Segment reference .....	293
Implementation-defined behavior for Standard C .....	311
Implementation-defined behavior for C89 .....	327
Index .....	341

# Contents

Tables .....	23
Preface .....	25
<b>Who should read this guide</b> .....	25
<b>How to use this guide</b> .....	25
<b>What this guide contains</b> .....	25
Part 1. Using the compiler .....	26
Part 2. Reference information .....	26
<b>Other documentation</b> .....	27
User and reference guides .....	27
The online help system .....	28
Further reading .....	28
Web sites .....	29
<b>Document conventions</b> .....	29
Typographic conventions .....	30
Naming conventions .....	30
<b>Part I. Using the compiler</b> .....	33
Getting started .....	35
<b>IAR language overview</b> .....	35
<b>Supported CRI6C devices</b> .....	36
<b>Building applications—an overview</b> .....	36
Compiling .....	36
Linking .....	37
<b>Basic project configuration</b> .....	37
Data model .....	38
Code model .....	38
Optimization for speed and size .....	39
Runtime environment .....	39
<b>Special support for embedded systems</b> .....	41
Extended keywords .....	41

Pragma directives .....	41
Predefined symbols .....	41
Special function types .....	41
Accessing low-level features .....	42
<b>Data storage</b> .....	43
<b>Introduction</b> .....	43
Different ways to store data .....	43
<b>Data models</b> .....	44
Specifying a data model .....	44
<b>Memory types</b> .....	46
Short direct addressing .....	46
Long direct addressing .....	46
Indexed addressing .....	47
Using data memory attributes .....	47
Pointers and memory types .....	49
Structures and memory types .....	50
More examples .....	50
<b>C++ and memory types</b> .....	51
<b>Auto variables—on the stack</b> .....	52
The stack .....	52
<b>Dynamic memory on the heap</b> .....	53
Potential problems .....	53
<b>Functions</b> .....	55
<b>Function-related extensions</b> .....	55
<b>Code models and register modes</b> .....	55
<b>Primitives for interrupts, concurrency, and OS-related programming</b> .....	56
Interrupt functions .....	56
Trap functions .....	57
User functions .....	58
Monitor functions .....	58
C++ and special function types .....	62

<b>Inlining functions</b> .....	63
C versus C++ semantics .....	63
Features controlling function inlining .....	64
<b>Placing code and data</b> .....	65
<b>Segments and memory</b> .....	65
What is a segment? .....	65
<b>Placing segments in memory</b> .....	66
Customizing the linker configuration file .....	66
<b>Data segments</b> .....	69
Static memory segments .....	69
The stack .....	72
The heap .....	73
Located data .....	75
User-defined segments .....	75
<b>Code segments</b> .....	75
Startup code .....	75
Normal code .....	76
Interrupt vectors .....	76
<b>C++ dynamic initialization</b> .....	76
<b>Verifying the linked result of code and data placement</b> .....	76
Segment too long errors and range errors .....	76
Linker map file .....	77
<b>The DLIB runtime environment</b> .....	79
<b>Introduction to the runtime environment</b> .....	79
Runtime environment functionality .....	79
Setting up the runtime environment .....	80
<b>Using a prebuilt library</b> .....	81
Choosing a library .....	82
Library filename syntax .....	83
Customizing a prebuilt library without rebuilding .....	83
<b>Choosing formatters for printf and scanf</b> .....	84
Choosing a printf formatter .....	84
Choosing a scanf formatter .....	85

<b>Application debug support</b> .....	86
Including C-SPY debugging support .....	86
The debug library functionality .....	87
The C-SPY Terminal I/O window .....	88
Low-level functions in the debug library .....	89
<b>Adapting the library for target hardware</b> .....	89
Library low-level interface .....	90
<b>Overriding library modules</b> .....	90
<b>Building and using a customized library</b> .....	91
Setting up a library project .....	91
Modifying the library functionality .....	92
Using a customized library .....	92
<b>System startup and termination</b> .....	92
System startup .....	93
System termination .....	94
<b>Customizing system initialization</b> .....	95
__low_level_init .....	96
Modifying the file cstartup.s45 .....	96
<b>Library configurations</b> .....	96
Choosing a runtime configuration .....	97
<b>Standard streams for input and output</b> .....	97
Implementing low-level character input and output .....	98
<b>Configuration symbols for printf and scanf</b> .....	100
Customizing formatting capabilities .....	101
<b>File input and output</b> .....	102
<b>Locale</b> .....	102
Locale support in prebuilt libraries .....	103
Customizing the locale support .....	103
Changing locales at runtime .....	104
<b>Environment interaction</b> .....	105
The getenv function .....	105
The system function .....	105
<b>Signal and raise</b> .....	106
<b>Time</b> .....	106



<b>Strtod</b> .....	107
<b>Math functions</b> .....	107
Smaller versions .....	107
More accurate versions .....	108
<b>Assert</b> .....	109
<b>Heaps</b> .....	109
<b>Checking module consistency</b> .....	110
Runtime model attributes .....	110
Using runtime model attributes .....	111
Predefined runtime attributes .....	111
<b>The CLIB runtime environment</b> .....	113
<b>Using a prebuilt library</b> .....	113
Choosing a library .....	114
<b>Input and output</b> .....	115
Character-based I/O .....	115
Formatters used by printf and sprintf .....	115
Formatters used by scanf and sscanf .....	117
<b>System startup and termination</b> .....	117
System startup .....	118
System termination .....	118
<b>Overriding default library modules</b> .....	118
<b>Customizing system initialization</b> .....	119
<b>C-SPY runtime interface</b> .....	119
The debugger Terminal I/O window .....	119
Termination .....	119
<b>Checking module consistency</b> .....	119
<b>Assembler language interface</b> .....	121
<b>Mixing C and assembler</b> .....	121
Intrinsic functions .....	121
Mixing C and assembler modules .....	122
Inline assembler .....	123
<b>Calling assembler routines from C</b> .....	125
Creating skeleton code .....	125

Compiling the code .....	126
<b>Calling assembler routines from C++ .....</b>	<b>127</b>
<b>Calling convention .....</b>	<b>128</b>
Function declarations .....	128
Using C linkage in C++ source code .....	128
Preserved versus scratch registers .....	129
Function entrance .....	130
Function exit .....	131
Restrictions for special function types .....	133
Examples .....	133
Function directives .....	134
<b>Calling functions .....</b>	<b>135</b>
Assembler instructions used for calling functions .....	135
<b>Memory access methods .....</b>	<b>136</b>
The data16 memory access method .....	136
The data20 memory access method .....	137
The data24 memory access method .....	137
The data32 memory access method .....	138
The index20 memory access method .....	138
The index4 and index5 memory access methods .....	139
<b>Call frame information .....</b>	<b>139</b>
CFI directives .....	139
Creating assembler source with CFI support .....	140
<b>Using C .....</b>	<b>145</b>
<b>C language overview .....</b>	<b>145</b>
<b>Extensions overview .....</b>	<b>146</b>
Enabling language extensions .....	147
<b>IAR C language extensions .....</b>	<b>147</b>
Extensions for embedded systems programming .....	148
Relaxations to Standard C .....	150
<b>Using C++ .....</b>	<b>153</b>
<b>Overview .....</b>	<b>153</b>
Embedded C++ .....	153

Extended Embedded C++ .....	154
<b>Enabling support for C++ .....</b>	<b>154</b>
<b>EC++ feature descriptions .....</b>	<b>155</b>
Using IAR attributes with Classes .....	155
Function types .....	158
New and Delete operators .....	159
Using static class objects in interrupts .....	160
Using New handlers .....	161
Templates .....	161
Debug support in C-SPY .....	161
<b>EEC++ feature description .....</b>	<b>161</b>
Templates .....	161
Variants of cast operators .....	165
Mutable .....	165
Namespace .....	165
The STD namespace .....	165
<b>C++ language extensions .....</b>	<b>166</b>
<b>Efficient coding for embedded applications .....</b>	<b>169</b>
<b>Selecting data types .....</b>	<b>169</b>
Using efficient data types .....	169
Floating-point types .....	170
Alignment of elements in a structure .....	171
Anonymous structs and unions .....	171
<b>Controlling data and function placement in memory .....</b>	<b>173</b>
Data placement at an absolute location .....	174
Data and function placement in segments .....	175
<b>Controlling compiler optimizations .....</b>	<b>177</b>
Scope for performed optimizations .....	177
Multi-file compilation units .....	177
Optimization levels .....	178
Speed versus size .....	179
Fine-tuning enabled transformations .....	179

<b>Facilitating good code generation</b> .....	181
Writing optimization-friendly source code .....	182
Saving stack space and RAM memory .....	182
Function prototypes .....	182
Integer types and bit negation .....	183
Protecting simultaneously accessed variables .....	184
Accessing special function registers .....	185
Non-initialized variables .....	185
<b>Part 2. Reference information</b> .....	187
<b>External interface details</b> .....	189
<b>Invocation syntax</b> .....	189
Compiler invocation syntax .....	189
Passing options .....	189
Environment variables .....	190
<b>Include file search procedure</b> .....	190
<b>Compiler output</b> .....	191
Error return codes .....	192
<b>Diagnostics</b> .....	192
Message format .....	193
Severity levels .....	193
Setting the severity level .....	194
Internal error .....	194
<b>Compiler options</b> .....	195
<b>Options syntax</b> .....	195
Types of options .....	195
Rules for specifying parameters .....	195
<b>Summary of compiler options</b> .....	197
<b>Descriptions of compiler options</b> .....	200
--c89 .....	200
--char_is_signed .....	201
--char_is_unsigned .....	201

--clib .....	201
--code_model .....	202
-D .....	202
--data_model .....	203
--debug, -r .....	203
--dependencies .....	203
--diag_error .....	204
--diag_remark .....	205
--diag_suppress .....	205
--diag_warning .....	206
--diagnostics_tables .....	206
--discard_unused_publics .....	206
--dlib .....	207
--dlib_config .....	207
-e .....	208
--ec++ .....	208
--eec++ .....	209
--enable_multibytes .....	209
--error_limit .....	209
-f .....	210
--guard_calls .....	210
--header_context .....	210
-I .....	211
-l .....	211
--library_module .....	212
--macro_positions_in_diagnostics .....	212
--mfc .....	213
--module_name .....	213
--no_code_motion .....	213
--no_cse .....	214
--no_inline .....	214
--no_path_in_file_macros .....	214
--no_size_constraints .....	215
--no_static_destruction .....	215

--no_system_include .....	215
--no_tbaa .....	216
--no_typedefs_in_diagnostics .....	216
--no_unroll .....	216
--no_warnings .....	217
--no_wrap_diagnostics .....	217
-O .....	217
--omit_types .....	218
--only_stdout .....	218
--output, -o .....	218
--predef_macros .....	219
--preinclude .....	219
--preprocess .....	220
--public_equ .....	220
--relaxed_fp .....	220
--remarks .....	221
--require_prototypes .....	221
--segment .....	222
--silent .....	223
--strict .....	223
--system_include_dir .....	224
--use_c++_inline .....	224
--vla .....	224
--warnings_affect_exit_code .....	225
--warnings_are_errors .....	225

Data representation .....	227
<b>Alignment</b> .....	227
Alignment on the CR16C microprocessor .....	227
<b>Basic data types</b> .....	228
Integer types .....	228
Floating-point types .....	231
<b>Pointer types</b> .....	233
Size .....	233

Casting .....	233
Alignment .....	234
General layout .....	234
Packed structure types .....	235
<b>Type qualifiers</b> .....	236
Declaring objects volatile .....	236
Declaring objects volatile and const .....	237
Declaring objects const .....	238
<b>Data types in C++</b> .....	238
Extended keywords .....	239
<b>General syntax rules for extended keywords</b> .....	239
Type attributes .....	239
Object attributes .....	242
<b>Summary of extended keywords</b> .....	242
<b>Descriptions of extended keywords</b> .....	243
__data16 .....	243
__data20 .....	244
__data24 .....	244
__data32 .....	245
__interrupt .....	245
__intrinsic .....	246
__ix4 .....	246
__ix20 .....	246
__monitor .....	247
__noadjust .....	247
__no_init .....	247
__noreturn .....	248
__raw .....	248
__root .....	248
__task .....	248
__trap .....	249
__user .....	249

Pragma directives .....	251
<b>Summary of pragma directives</b> .....	251
<b>Descriptions of pragma directives</b> .....	252
basic_template_matching .....	252
bitfields .....	253
data_alignment .....	253
diag_default .....	254
diag_error .....	254
diag_remark .....	255
diag_suppress .....	255
diag_warning .....	255
error .....	256
include_alias .....	256
inline .....	257
language .....	257
location .....	258
message .....	259
object_attribute .....	259
optimize .....	260
pack .....	261
__printf_args .....	262
required .....	262
rtmodel .....	263
__scanf_args .....	263
segment .....	264
STDC CX_LIMITED_RANGE .....	265
STDC FENV_ACCESS .....	265
STDC FP_CONTRACT .....	265
type_attribute .....	266
vector .....	266



Intrinsic functions .....	269
<b>Summary of intrinsic functions</b> .....	269
<b>Descriptions of intrinsic functions</b> .....	270
__adjust_return_address .....	270
__clear_processor_register_bit .....	270
__clear_PSR_I_bit .....	271
__disable_interrupt .....	271
__enable_interrupt .....	271
__enable_interrupt_wait .....	271
__get_interrupt_state .....	271
__get_processor_register .....	272
__mac_q15 .....	272
__mac_signed .....	273
__mac_unsigned .....	273
__memcpy_generic .....	273
__memset_generic .....	273
__no_operation .....	274
__raise_exception .....	274
__set_interrupt_state .....	274
__set_processor_register .....	274
__set_processor_register_bit .....	274
__set_PSR_I_bit .....	275
__wait_for_interrupt .....	275
The preprocessor .....	277
<b>Overview of the preprocessor</b> .....	277
<b>Description of predefined preprocessor symbols</b> .....	278
__BASE_FILE__ .....	278
__BUILD_NUMBER__ .....	278
__CODE_MODEL__ .....	278
__cplusplus__ .....	278
__DATA_MODEL__ .....	278
__DATE__ .....	279
__DOUBLE__ .....	279

__embedded_cplusplus .....	279
__FILE__ .....	279
__func__ .....	279
__FUNCTION__ .....	280
__IAR_SYSTEMS_ICC__ .....	280
__ICC_CR16C__ .....	280
__INDEXED_ENABLED__ .....	280
__LINE__ .....	280
__LITTLE_ENDIAN__ .....	280
__PRETTY_FUNCTION__ .....	280
__STDC__ .....	281
__STDC_VERSION__ .....	281
__SUBVERSION__ .....	281
__TIME__ .....	281
__VER__ .....	281
<b>Descriptions of miscellaneous preprocessor extensions</b> .....	282
NDEBUG .....	282
#warning message .....	282
<b>Library functions</b> .....	283
<b>Library overview</b> .....	283
Header files .....	283
Library object files .....	284
Alternative more accurate library functions .....	284
Reentrancy .....	284
The longjmp function .....	285
<b>IAR DLIB Library</b> .....	285
C header files .....	285
C++ header files .....	286
Library functions as intrinsic functions .....	289
Added C functionality .....	289
Symbols used internally by the library .....	290
<b>IAR CLIB Library</b> .....	291
Library definitions summary .....	291

Segment reference .....	293
<b>Summary of segments</b> .....	293
<b>Descriptions of segments</b> .....	295
ABSOLUTE_C .....	295
ABSOLUTE_N .....	295
CHECKSUM .....	296
CODE .....	296
CSTACK .....	296
CSTART .....	296
DATA16_C .....	297
DATA16_HEAP .....	297
DATA16_I .....	297
DATA16_ID .....	298
DATA16_N .....	298
DATA16_Z .....	298
DATA20_C .....	299
DATA20_I .....	299
DATA20_ID .....	299
DATA20_N .....	300
DATA20_Z .....	300
DATA24_C .....	300
DATA24_I .....	300
DATA24_ID .....	301
DATA24_N .....	301
DATA24_Z .....	301
DATA32_C .....	302
DATA32_HEAP .....	302
DATA32_I .....	302
DATA32_ID .....	303
DATA32_N .....	303
DATA32_Z .....	303
DIFUNCT .....	304
IDINIT .....	304

INDEX_BASE .....	304
INDEX4_I .....	304
INDEX4_ID .....	305
INDEX4_N .....	305
INDEX4_Z .....	305
INDEX5_I .....	306
INDEX5_ID .....	306
INDEX5_N .....	307
INDEX5_Z .....	307
INDEX20_C .....	308
INDEX20_I .....	308
INDEX20_ID .....	308
INDEX20_N .....	309
INDEX20_Z .....	309
INTVEC .....	309
ISTACK .....	309
ZINIT .....	310
<b>Implementation-defined behavior for Standard C .....</b>	<b>311</b>
<b>Descriptions of implementation-defined behavior .....</b>	<b>311</b>
J.3.1 Translation .....	311
J.3.2 Environment .....	312
J.3.3 Identifiers .....	313
J.3.4 Characters .....	313
J.3.5 Integers .....	315
J.3.6 Floating point .....	315
J.3.7 Arrays and pointers .....	316
J.3.8 Hints .....	317
J.3.9 Structures, unions, enumerations, and bitfields .....	317
J.3.10 Qualifiers .....	318
J.3.11 Preprocessing directives .....	318
J.3.12 Library functions .....	320
J.3.13 Architecture .....	324
J.4 Locale .....	325

Implementation-defined behavior for C89 .....	327
<b>Descriptions of implementation-defined behavior</b> .....	327
Translation .....	327
Environment .....	327
Identifiers .....	328
Characters .....	328
Integers .....	329
Floating point .....	330
Arrays and pointers .....	331
Registers .....	331
Structures, unions, enumerations, and bitfields .....	331
Qualifiers .....	332
Declarators .....	332
Statements .....	332
Preprocessing directives .....	332
IAR DLIB Library functions .....	334
IAR CLIB Library functions .....	337
<b>Index</b> .....	341



# Tables

1: Typographic conventions used in this guide .....	30
2: Naming conventions used in this guide .....	30
3: Data model characteristics .....	44
4: Memory types and their corresponding memory attributes .....	48
5: XLINK segment memory types .....	66
6: Memory layout of a target system (example) .....	67
7: Memory types with corresponding segment groups .....	70
8: Segment name suffixes .....	70
9: Prebuilt libraries .....	82
10: Customizable items .....	83
11: Formatters for printf .....	84
12: Formatters for scanf .....	85
13: Levels of debugging support in runtime libraries .....	86
14: Functions with special meanings when linked with debug library .....	89
15: Library configurations .....	97
16: Descriptions of printf configuration symbols .....	101
17: Descriptions of scanf configuration symbols .....	101
18: Low-level I/O files .....	102
19: Heaps and memory types .....	109
20: Example of runtime model attributes .....	110
21: Predefined runtime model attributes .....	111
22: Runtime libraries .....	114
23: Registers used for passing parameters .....	130
24: Registers used for returning values .....	132
25: Specifying the size of an assembler memory instruction .....	136
26: Call frame information resources defined in a names block .....	140
27: Language extensions .....	147
28: Compiler optimization levels .....	178
29: Compiler environment variables .....	190
30: Error return codes .....	192
31: Compiler options summary .....	197

32: Integer types .....	228
33: Floating-point types .....	231
34: Extended keywords summary .....	242
35: Pragma directives summary .....	251
36: Intrinsic functions summary .....	269
37: Traditional Standard C header files—DLIB .....	285
38: C++ header files .....	287
39: <Standard template library header files .....	287
40: New Standard C header files—DLIB .....	288
41: IAR CLIB Library header files .....	291
42: Segment summary .....	293
43: Message returned by strerror()—IAR DLIB library .....	326
44: Message returned by strerror()—IAR DLIB library .....	337
45: Message returned by strerror()—IAR CLIB library .....	340



# Preface

Welcome to the *IAR C/C++ Compiler Reference Guide for CR16C*. The purpose of this guide is to provide you with detailed reference information that can help you to use the compiler 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 CR16C microprocessor and need detailed reference information on how to use the compiler. You should have working knowledge of:

- The architecture and instruction set of the CR16C microprocessor. Refer to the documentation from the chip manufacturer for information about the CR16C microprocessor
- 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 for CR16C, you should read *Part 1. Using the compiler* in this guide.

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

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

---

## What this guide contains

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

## PART 1. USING THE COMPILER

- *Getting started* gives the information you need to get started using the compiler for efficiently developing your application.
- *Data storage* describes how to store data in memory, focusing on the different data models and data memory type attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker configuration file, and describes how code and data are placed in memory.
- *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.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how to customize them. The chapter also describes system initialization and introduces the file `cstartup`.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *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 interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler’s diagnostic system works.
- *External interface details* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Extended keywords* gives reference information about each of the CR16C-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 CR16C-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.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior for Standard C* describes how the compiler handles the implementation-defined areas of Standard C.
- *Implementation-defined behavior for C89* describes how the compiler handles the implementation-defined areas of the C language standard C89.

---

## Other documentation

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

### USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, refer to the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, see the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, see the *IDE Project Management and Building Guide*.
- Using the IAR C-SPY® Debugger, see the *C-SPY® Debugging Guide*.
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, see the IAR Linker and Library Tools Reference Guide.
- Programming for the IAR Assembler for CR16C, see the *IAR Assembler Reference Guide for CR16C*.

- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, see the *IAR C Library Functions Reference Guide*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for CR16C, see the *IAR Embedded Workbench® Migration Guide*.
- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
- Using the IAR C-SPY® Nexus Debugger Systems for CR16C, refer to the *IAR C-SPY® Nexus Debugger Systems User Guide for CR16C*.

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

## THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about debugging using the IAR C-SPY® Debugger
- Information about using the editor
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1. Note that if you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

## FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall.

- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.
- Meyers, Scott. *Effective STL*. Addison-Wesley.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++*. Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.

## WEB SITES

Recommended web sites:

- The IAR Systems web site, [www.iar.com](http://www.iar.com), that holds application notes and other product information.
- The web site of the C standardization working group, [www.open-std.org/jtc1/sc22/wg14](http://www.open-std.org/jtc1/sc22/wg14).
- The web site of the C++ Standards Committee, [www.open-std.org/jtc1/sc22/wg21](http://www.open-std.org/jtc1/sc22/wg21).
- Finally, the Embedded C++ Technical Committee web site, [www.caravan.net/ec2plus](http://www.caravan.net/ec2plus), that contains information about the Embedded C++ standard.

---

## Document conventions

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

When referring to a directory in your product installation, for example `cr16c\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.n\cr16c\doc`.

## TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:





Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> <li>• Source code examples and file paths.</li> <li>• Text on the command line.</li> <li>• Binary, hexadecimal, and octal numbers.</li> </ul>
<code>parameter</code>	A placeholder for an actual value used as a parameter, for example <code>filename.h</code> where <code>filename</code> represents the name of the file.
<code>[option]</code>	An optional part of a command.
<code>[a b c]</code>	An optional part of a command with alternatives.
<code>{a b c}</code>	A mandatory part of a command with alternatives.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> <li>• A cross-reference within this guide or to another guide.</li> <li>• Emphasis.</li> </ul>
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

## NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for CR16C	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for CR16C	the IDE
IAR C-SPY® Debugger for CR16C	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for CR16C	the compiler

Table 2: Naming conventions used in this guide

<b>Brand name</b>	<b>Generic term</b>
IAR Assembler™ for CR16C	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

*Table 2: Naming conventions used in this guide (Continued)*

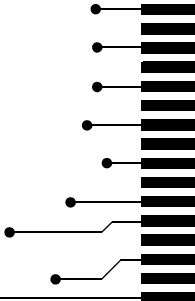


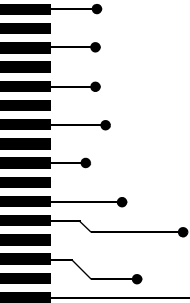


# Part I. Using the compiler

This part of the *IAR C/C++ Compiler Reference Guide for CR16C* includes these chapters:

- Getting started
- Data storage
- Functions
- Placing code and data
- The DLIB runtime environment
- The CLIB runtime environment
- Assembler language interface
- Using C
- Using C++
- Efficient coding for embedded applications.





# Getting started

This chapter gives the information you need to get started using the compiler for efficiently developing your application.

First you will get an overview of the supported programming languages, followed by a description of the steps involved for compiling and linking an application.

Next, the compiler is introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the CR16C microprocessor. In the following chapters, these techniques are studied in more detail.

---

## IAR language overview

There are two high-level programming languages you can use with the IAR C/C++ Compiler for CR16C

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

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

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

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

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *IAR Assembler Reference Guide for CR16C*.

---

## Supported CR16C devices

The IAR C/C++ Compiler for CR16C supports all devices based on the standard CR16C microprocessor. The following extensions are also supported:

- The CR16CPlus core, including support for real-time code profiling.
- These SiTel co-processors: SC14428, SC14429, SC14430, SC14434, SC14438, SC14450, SC14451, SC14470, SC14471, and SC14480.

---

## Building applications—an overview

A typical application is built from several source files and libraries. The source files can be written in C, C++, or assembler language, and can be compiled into object files by the compiler or the assembler.

A library is a collection of object files that are added at link time only if they are needed. A typical example of a library is the compiler library containing the runtime environment and the C/C++ standard library. Libraries can also be built using the IAR XAR Library Builder, the IAR XLIB Librarian, or be provided by external suppliers.

The IAR XLINK Linker is used for building the final application. XLINK normally uses a linker configuration file, which describes the available resources of the target system.



Below, the process for building an application on the command line is described. For information about how to build an application using the IDE, see the *IDE Project Management and Building Guide*.

### COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r45` using the default settings:

```
icccr16c myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 37.

## LINKING

The IAR XLINK Linker is used for building the final application. Normally, XLINK requires the following information as input:

- One or more object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label
- A linker configuration file that describes the placement of code and data into the memory of the target system
- Information about the output format.

On the command line, the following line can be used for starting XLINK:

```
xlink myfile.r45 myfile2.r45 -s __program_start -f lnkcr16c.xcl
clcr16cns.r45 -o aout.a45 -r
```

In this example, `myfile.r45` and `myfile2.r45` are object files, `lnkcr16c.xcl` is the linker configuration file, and `clcr16cns.r45` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

The IAR XLINK Linker produces output according to your specifications. Choose the output format that suits your purpose. You might want to load the output to a debugger—which means that you need output with debug information. Alternatively, you might want to load the output to a flash loader or a PROM programmer—in which case you need output without debug information, such as Intel hex or Motorola S-records. The option `-F` can be used for specifying the output format. (The default output format is `motorola`.)

---

## Basic project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler and linker generate the best code for the CR16C device you are using. You can specify the options either from the command line interface or in the IDE.

You need to make settings for:

- Data model
- Code model
- Optimization settings
- Runtime environment

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

## DATA MODEL

One of the characteristics of the CR16C microprocessor is a trade-off in how memory is accessed, between the range from cheap access to small memory areas, up to more expensive access methods that can access any location.

In the compiler, you can set a default memory access method by selecting a data model. The following data models are supported:

- In the *small* data model, data including the stack must be placed in the first 64 Kbytes of the memory. The small data model is the default data model.
- In the *medium* data model, data is placed in the first 1 Mbyte of the memory.
- In the *large* data model, data is placed in 16 Mbytes of the memory.
- In the *huge* data model, data is placed anywhere in memory.
- In the *indexed* data model, data is placed anywhere in a 1-Mbyte area of the memory.

However, it is possible to override the default access method for each individual variable.

The chapter *Data storage* covers data models in greater detail. The chapter also covers how to fine-tune the access method for individual variables.

## CODE MODEL

The CR16C microprocessor can be used in two modes: normal mode and short register mode. The compiler supports these modes by means of code models. The code model controls how code is generated for an application. All object files of a system must be compiled using the same code model.

The compiler provides two code models:

- The *normal* code model, which is the default, supports the normal register mode
- The *short* code model supports the short register mode.

In the chapter *Assembler language interface*, the generated code is studied in more detail when we describe how to call a C function from assembler language and vice versa.

## OPTIMIZATION FOR SPEED AND SIZE

The compiler's optimizer performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

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

## RUNTIME ENVIRONMENT

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

Two different sets of runtime libraries are provided:

- The IAR DLIB Library, which supports Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc. (This library is used by default).
- The IAR CLIB Library is a light-weight library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++.

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

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

For more information about the runtime environments, see the chapters *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.



## Setting up for the runtime environment in the IDE

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

Note that for the DLIB library there are different configurations— Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, etc. See *Library configurations*, page 96, for more information.



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

On the compiler command line, specify whether you want the system header files for DLIB or CLIB by using the `--dlib` option or the `--clib` option. If you use the DLIB library, you can use the `--dlib_config` option instead if you also want to explicitly define which library configuration to be used.

On the linker command line, you must specify which runtime library object file to be used. The linker command line can for example look like this:

```
d1cr16cns.r45
```

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

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

```
-I MyApplication\inc
```

For information about the prebuilt library object files, see *Using a prebuilt library*, page 81 (DLIB) and *Using a prebuilt library*, page 113 (CLIB). 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 84 (DLIB) and *Input and output*, page 115 (CLIB).
- The size of the stack and the heap, see *The stack*, page 72, and *The heap*, page 73, respectively.



---

## Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the CR16C microprocessor.

### 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 controlling the memory type for individual variables as well as 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 208 for additional information.

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

### PRAGMA DIRECTIVES

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

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

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

### PREDEFINED SYMBOLS

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example the code and data models.

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

### SPECIAL FUNCTION TYPES

The special hardware features of the CR16C microprocessor are supported by the compiler's special function types: interrupt, monitor, user, task, and trap. You can write a complete application without having to write any of these functions in assembler language.

For more information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 56.

## **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 121.

# Data storage

This chapter gives a brief introduction to the memory layout of the CR16C microprocessor and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. For efficient memory usage, the compiler provides a set of data models and data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of data models and memory types are described in relation to pointers, structures, Embedded C++ class objects, and non-initialized memory. Finally, detailed information about data storage on the stack and the heap is provided.

---

## Introduction

The compiler supports CR16C devices with different sizes and layouts of continuous memory. 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.

The compiler can access memory in different ways. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. For more information about this, see *Memory types*, page 46.

### DIFFERENT WAYS TO STORE DATA

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

- Auto variables  
All variables that are local to a function, except those declared static, are stored either in registers or on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid. For more information, see *Auto variables—on the stack*, page 52.
- Global variables, module-static variables, and local variables declared `static`  
In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Data models*, page 44 and *Memory types*, page 46.

- 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 53.

---

## Data models

Use *data models* to specify in which part of memory the compiler should place static and global variables by default. This means that the data model controls:

- The default memory type
- The default placement of static and global variables, and constant literals
- Dynamically allocated data, for example data allocated with `malloc`, or, in C++, the operator `new`
- The default pointer type
- The placement of the runtime stack.

The data model only specifies the default memory type. It is possible to override this for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 47.

### SPECIFYING A DATA MODEL

Four data models are implemented: Small, Medium, Large, and Indexed. These models are controlled by the `--data_model` option. Each model has a default memory type and a default pointer size. If you do not specify a data model option, the compiler will use the Small data model.

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 47.

This table summarizes the different data models:

Data model name	Default memory attribute	Default pointer size	Placement of data
Small (default)	<code>__data16</code>	16	First 64 Kbytes

Table 3: Data model characteristics

Data model name	Default memory attribute	Default pointer size	Placement of data
Medium	<code>__data20</code>	32	First 1 Mbyte
Large	<code>__data24</code>	32	First 16 Mbytes
Huge	<code>__data32</code>	32	Anywhere in memory
Indexed	<code>__ix20</code>	32	1 Mbyte area, anywhere

Table 3: Data model characteristics (Continued)



See the *IDE Project Management and Building Guide* for information about setting options in the IDE.



Use the `--data_model` option to specify the data model for your project; see *--data\_model*, page 203.

### The Small data model

The Small data model places data in the first 64 Kbytes of memory. The default memory used in the Small data model can be accessed using 16-bit pointers. The advantage is that only 16 bits are needed for pointer storage. The default pointer type passed as a parameter will use one register or 2 byte on the stack.

### The Medium data model

The Medium data model places data in the first 1 Mbyte of memory. The default memory used in the Medium data model can be accessed using 32-bit pointers. The default pointer type passed as a parameter will use one register or 4 bytes on the stack.

### The Large data model

The Large data model places data in the first 16 Mbytes of memory. The default memory used in the Large data model can be accessed using 32-bit pointers. The default pointer type passed as a parameter will use one register or 4 bytes on the stack.

### The Huge data model

The Huge data model places data anywhere in memory. The default memory used in the Huge data model can be accessed using 32-bit pointers. The default pointer type passed as a parameter will use one register or 4 bytes on the stack.

### The Indexed data model

The Indexed data model places data in a 1-Mbyte area of memory. This area can be placed anywhere in memory. The default memory used in the Indexed data model can be accessed using 32-bit pointers. The default pointer type passed as a parameter will use one register or 4 bytes on the stack.

---

## Memory types

This section describes the concept of *memory types* used for accessing data by the compiler. It also discusses pointers in the presence of multiple memory types. For each memory type, the capabilities and limitations are discussed.

The compiler uses different memory types to access data that is placed in different areas of the memory. There are different methods for reaching memory areas, and they have different costs when it comes to code space, execution speed, and register usage. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. Each memory type corresponds to one memory access method. If you map different memories—or part of memories—to memory types, the compiler can generate code that can access data efficiently.

For example, the memory accessed using 16-bit addressing is called data16 memory.

To choose a default memory type that your application will use, select a *data model*. However, it is possible to specify—for individual variables or pointers—different memory types. This makes it possible to create an application that can contain a large amount of data, and at the same time make sure that variables that are used often are placed in memory that can be efficiently accessed.

For more information about memory access methods, see *Memory access methods*, page 136.

### SHORT DIRECT ADDRESSING

Short direct addressing gives access to a smaller memory area than long direct addressing.

#### Data16

The data16 memory consists of the low 64 Kbytes of data memory. In hexadecimal notation, this is the address range 0x0000–0xFFFF.

The size of a data16 object is limited to 32 Kbytes. If you use objects of this type, the code generated by the compiler to access them is minimized. This means a smaller footprint for the application, and faster execution at runtime.

### LONG DIRECT ADDRESSING

Long direct addressing gives access to a larger memory area than short direct addressing. However, the drawback of large direct addressing is that the code generated to access the memory is larger and slower than that of short direct addressing. The code also uses more processor registers, which might force local variables to be stored on the stack rather than being allocated in registers.

**Data20**

Data20 objects must be placed in the first 1 Mbyte of memory.

**Data24**

Data24 objects must be placed in the first 16 Mbyte of memory.

**Data32**

Data32 objects can be placed anywhere in data memory.

In fact, the processor architecture does not support direct memory access to this memory type, which requires more complex code than for any other memory type. The data32 memory type is primarily useful for CR16CPlus.

**INDEXED ADDRESSING**

The name indexed comes from the indexed addressing mode with the processor register R13 as a base pointer. The base pointer can be placed anywhere in memory, and the indexed memory types are relative to the base pointer.

The index4 and index5 memory access methods are more efficient than the index20 method, and should be used for variables that are accessed often.

**Index4**

The index4 access method can access the first 14 bytes following the base pointer.

**Index5**

The index5 access method can access 28 bytes following the base pointer. It is not possible, however, to access single bytes. A variable declared using the `__ix4` keyword will be accessed using the index5 method if it is a scalar larger than a character or a structure without `char` members.

**Index20**

Using index20 addressing, a 1-Mbyte area of the memory can be accessed.

**USING DATA MEMORY ATTRIBUTES**

The compiler provides a set of *extended keywords*, which can be used as *data memory attributes*. These keywords let you override the default memory type for individual data objects and pointers, which means that you can place data objects in other memory areas than the default memory. This also means that you can fine-tune the access method for each individual data object, which results in smaller code size.

This table summarizes the available memory types and their corresponding keywords:

Memory type	Memory attribute	Address range	Pointer attribute	Pointer size	Default in data model
Data16	<code>__data16</code>	0x0-0xFFFF	<code>__data16</code>	16 bits	Small
Data20	<code>__data20</code>	0x0-0xFFFFF	<code>__data32</code>	32 bits	Medium
Data24	<code>__data24</code>	0x0-0xFFFFF	<code>__data32</code>	32 bits	Large
Data32	<code>__data32</code>	0x0-0xFFFFFFFF	<code>__data32</code>	32 bits	Huge
Index4	<code>__ix4</code>	<i>base to base + 14 bytes*</i>	<code>__data32</code>	32 bits	—
Index5	<code>__ix4</code>	<i>base to base + 28 bytes*</i>	<code>__data32</code>	32 bits	—
Index20	<code>__ix20</code>	<i>base to base + 0xFFFFF*</i>	<code>__data32</code>	32 bits	Indexed

Table 4: Memory types and their corresponding memory attributes

\* *base* can be placed anywhere in memory but is the same location for `index4`, `index5`, and `index20`.

The keywords are only available if 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 208 for additional information.

For backward compatibility, the keywords `__near` and `__sbrl` are available as aliases for `__data16` and `__ix4`.

For more information about each keyword, see *Descriptions of extended keywords*, page 243.

## Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when variables are defined and in the declaration, see *General syntax rules for extended keywords*, page 239.



The following declarations place the variables `i` and `j` in `data20` memory. The variables `k` and `l` will also be placed in `data20` memory. The position of the keyword does not have any effect in this case:

```
__data20 int i, j;
int __data20 k, l;
```

Note that the keyword affects both identifiers. If no memory type is specified, the default memory type is used.

The `#pragma type_attribute` directive can also be used for specifying the memory attributes. 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. Refer to the chapter *Pragma directives* for details about how to use the extended keywords together with pragma directives.

## Type definitions

Storage can also be specified using type definitions. These two declarations are equivalent:

```
/* Defines via a typedef */
typedef char __data32 Byte;
typedef Byte *BytePtr;
Byte aByte;
BytePtr aBytePointer;

/* Defines directly */
__data32 char aByte;
char __data32 *aBytePointer;
```

## POINTERS AND MEMORY TYPES

Pointers are used for referring to the location of data. In general, a pointer has a type. For example, a pointer that has the type `int *` points to an integer.

In the compiler, a pointer also points to some type of memory. The memory type is specified using a keyword before the asterisk. For example, a pointer that points to an integer stored in `data32` memory is declared by:

```
int __data32 * MyPtr;
```

Note that the location of the pointer variable `MyPtr` is not affected by the keyword. In the following example, however, the pointer variable `MyPtr2` is placed in `data20` memory. Like `MyPtr`, `MyPtr2` points to a character in `data32` memory.

```
char __data32 * __data20 MyPtr2;
```

For example, the functions in the standard library are all declared without explicit memory types.

### Differences between pointer types

In the compiler, it is illegal to cast a 32-bit pointer to a 16-bit pointer.

Whenever possible, pointers should be declared without memory attributes. For example, the functions in the standard library are all declared without explicit memory types.

For more information about pointers, see *Pointer types*, page 233.

## STRUCTURES AND MEMORY TYPES

For structures, the entire object is placed in the same memory type. It is not possible to place individual structure members in different memory types.

In the example below, the variable `gamma` is a structure placed in `data20` memory.

```
struct MyStruct
{
    int mAlpha;
    int mBeta;
};

__data20 struct MyStruct gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
    int mAlpha;
    __data20 int mBeta; /* Incorrect declaration */
};
```

### MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in `data16` memory is declared. The function returns a pointer to an integer

in data20 memory. It makes no difference whether the memory attribute is placed before or after the data type.

<code>int MyA;</code>	A variable defined in default memory determined by the data model in use.
<code>int __data16 MyB;</code>	A variable in data16 memory.
<code>__data20 int MyC;</code>	A variable in data20 memory.
<code>int * MyD;</code>	A pointer stored in default memory. The pointer points to an integer in default memory.
<code>int __data16 * MyE;</code>	A pointer stored in default memory. The pointer points to an integer in data16 memory.
<code>int __data16 * __data20 MyF;</code>	A pointer stored in data20 memory pointing to an integer stored in data16 memory.
<code>int __data32 * MyFunction( int __data16 *);</code>	A declaration of a function that takes a parameter which is a pointer to an integer stored in data16 memory. The function returns a pointer to an integer stored in data32 memory.

---

## C++ and memory types

Instances of C++ classes are placed into a memory (just like all other objects) either implicitly, or explicitly using memory type attributes or other IAR language extensions. Non-static member variables, like structure fields, are part of the larger object and cannot be placed individually into specified memories.

In non-static member functions, the non-static member variables of a C++ object can be referenced via the `this` pointer, explicitly or implicitly. The `this` pointer is of the default data pointer type unless class memory is used, see *Using IAR attributes with Classes*, page 155.

Static member variables can be placed individually into a data memory in the same way as free variables.

For more information about C++ classes, see *Using IAR attributes with Classes*, page 155.

---

## Auto variables—on the stack

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

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

### THE STACK

The stack can contain:

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

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

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

### Advantages

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

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

### Potential problems

The way the stack works makes it impossible to store data that is supposed to live after the function returns. The following function demonstrates a common programming

mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns.

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

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

---

## Dynamic memory on the heap

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

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

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

The compiler supports heaps in both `data16` memory and `data32` memory. For more information about this, see *The heap*, page 73.

### POTENTIAL PROBLEMS

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

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

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

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

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

# Functions

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

---

## Function-related extensions

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

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

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

For more information about optimizations, see *Efficient coding for embedded applications*, page 169. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

---

## Code models and register modes

The CR16C microprocessor can be used in two modes: the Normal register mode and the Short register mode. The compiler supports these modes by means of code models. The code model controls how code is generated for an application. All object files of a system must be compiled using the same code model.

The compiler provides two code models:

- The Normal code model, which is the default, supports the Normal register mode.  
In the Normal register mode, `R12`, `R13`, and `RA` are 32-bit registers. The register `RA` holds the return address of functions.
- The Short code model supports the Short register mode.  
In the Short register mode, `R12`, `R13`, and `RA` are 16-bit registers. The register pair (`RA`, `R13`) holds the return address. The Short register mode is designed to be compatible with the CR16B large model.

In the chapter *Assembler language interface*, the generated code is studied in more detail when we describe how to call a C function from assembler language and vice versa.

---

## Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for CR16C provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__interrupt`, `__task`, `__trap`, `__user`, and `__monitor`
- The pragma directive `#pragma vector`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, and `__set_interrupt_state`.

### INTERRUPT FUNCTIONS

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

#### Interrupt service routines

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

The CR16C microprocessor supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the CR16C microprocessor documentation from the chip manufacturer. If you want to handle several different interrupts using the same interrupt routine, you can specify several interrupt vectors.

#### Interrupt vectors and the interrupt vector table

The start of the interrupt vector table is controlled by a special processor register, defined in the source file `cstartup.s45` (which you can find in the `cr16c\src\lib` directory). The table is placed in the `INTVEC` segment. The interrupt vector is the offset into the interrupt vector table.

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a



vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime.

The header file `iodevice.h`, where `device` corresponds to the selected device, contains predefined names for the existing interrupt vectors.

### Defining an interrupt function—an example

To define an interrupt function, the `__interrupt` keyword and the `#pragma vector` directive can be used. For example:

```
/* Symbol defined in I/O header file */
#pragma vector = UART_RI_INT
__interrupt void MyInterruptRoutine(void)
{
    /* Do something */
}
```

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

## TRAP FUNCTIONS

A trap is a kind of exception that can be activated when a specific event occurs or is called, by using the processor instruction `EXCP`. In many respects, a trap function behaves as a normal function; it can accept parameters, and return a value.

The typical use for trap functions is for the client interface of an operating system. If this interface is implemented using trap functions, the operating system part of an application can be updated independently of the rest of the system.

Each trap function is typically associated with a vector. The header file `iodevice.h`, which corresponds to the selected device, contains predefined names for the existing exception vectors.

The `__trap` keyword and the `#pragma vector` directive can be used to define trap functions. For example, this piece of code defines a function doubling its argument:

```
#pragma vector = __BPT
__trap int Twice(int x)
{
    return x + x;
}
```

When a trap function is defined with a vector, the processor interrupt vector table is populated. It is also possible to define a trap function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime. See the chip manufacturer's CR16C microprocessor documentation for more information about the interrupt vector table.

When a trap function is used, the compiler ensures that the application also will include the appropriate trap-handling code. See the chapter *Assembler language interface* for more information.

When trap functions are being called using the processor instruction `EXCP`, the return address will point to the instruction itself. To return to the instruction after the `EXCP` instruction, the return address will by default be adjusted within the trap function. Use the keyword `__noadjust` if you want to disable this adjustment. See also `__noadjust`, page 247.

## USER FUNCTIONS

The CR16C microprocessor supports two modes for executing code: supervisor mode and user mode. Supervisor mode is intended for use by system code—such as an operating system—and user mode for non-system code. In the latter case not all features of the processor are available. See `__user`, page 249.

The intended use of supervisor and user mode is for operating systems where the core will run in supervisor mode and user applications in user mode.

The compiler supports the declaration of user functions. When such functions are called, the processor switches to user mode. Note that when the call to the user function returns, the processor is still in user mode.

Also note that the processor uses different stacks in user and supervisor mode. This means that when calling functions that switch between the modes—such as user functions and possibly trap functions—the stack should not be used for passing parameters or return values. This is not checked by the compiler.

## MONITOR FUNCTIONS

A monitor function causes interrupts to be disabled during execution of the function. At function entry, the status register is saved and interrupts are disabled. At function exit, the original status register is restored, and thereby the interrupt status that existed before the function call is also restored.

To define a monitor function, you can use the `__monitor` keyword. For more information, see `__monitor`, page 247.



Avoid using the `__monitor` keyword on large functions, since the interrupt will otherwise be turned off for too long.

## Example of implementing a semaphore in C

In the following example, a binary semaphore—that is, a mutex—is implemented using one static variable and two monitor functions. A monitor function works like a critical region, that is no interrupt can occur and the process itself cannot be swapped out. A

semaphore can be locked by one process, and is used for preventing processes from simultaneously using resources that can only be used by one process at a time, for example a USART. The `__monitor` keyword assures that the lock operation is atomic; in other words it cannot be interrupted.

```
/* This is the lock-variable. When non-zero, someone owns it. */
static volatile unsigned int sTheLock = 0;
```

```
/* Function to test whether the lock is open, and if so take it.
 * Returns 1 on success and 0 on failure.
 */
```

```
__monitor int TryGetLock(void)
{
    if (sTheLock == 0)
    {
        /* Success, nobody has the lock. */

        sTheLock = 1;
        return 1;
    }
    else
    {
        /* Failure, someone else has the lock. */

        return 0;
    }
}
```

```
/* Function to unlock the lock.
 * It is only callable by one that has the lock.
 */
```

```
__monitor void ReleaseLock(void)
{
    sTheLock = 0;
}
```

```

/* Function to take the lock. It will wait until it gets it. */

void GetLock(void)
{
    while (!TryGetLock())
    {
        /* Normally, a sleep instruction is used here. */
    }
}

/* An example of using the semaphore. */

void MyProgram(void)
{
    GetLock();

    /* Do something here. */

    ReleaseLock();
}

```

### Example of implementing a semaphore in C++

In C++, it is common to implement small methods with the intention that they should be inlined. However, the compiler does not support inlining of functions and methods that are declared using the `__monitor` keyword.

In the following example in C++, an auto object is used for controlling the monitor block, which uses intrinsic functions instead of the `__monitor` keyword.

```
#include <intrinsics.h>

// Class for controlling critical blocks.

class Mutex
{
public:
    Mutex()
    {
        // Get hold of current interrupt state.
        mState = __get_interrupt_state();

        // Disable all interrupts.
        __disable_interrupt();
    }

    ~Mutex()
    {
        // Restore the interrupt state.
        __set_interrupt_state(mState);
    }

private:
    __istate_t mState;
};
```

```

class Tick
{
public:
    // Function to read the tick count safely.
    static long GetTick()
    {
        long t;

        // Enter a critical block.
        {
            Mutex m; // Interrupts are disabled while m is in scope.

            // Get the tick count safely,
            t = smTickCount;
        }
        // and return it.
        return t;
    }

private:
    static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
    static long nextStop = 100;

    if (Tick::GetTick() >= nextStop)
    {
        nextStop += 100;
        DoStuff();
    }
}

```

## C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, two restrictions apply:

- 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.

- Trap and user member functions cannot be declared virtual. The reason for this is that they cannot be called via function pointers.

---

## Inlining functions

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

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed. Normally, code size does not increase when optimizing for size.

### C VERSUS C++ SEMANTICS

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

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

For example:

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

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

## FEATURES CONTROLLING FUNCTION INLINING

There are several mechanisms for controlling function inlining:

- The `inline` keyword advises the compiler that the function defined immediately after the directive should be inlined.

If you compile your function in C or C++ mode, the keyword will be interpreted according to its definition in Standard C or Standard C++, respectively.

The main difference in semantics is that in Standard C you cannot (in general) simply supply an inline definition in a header file. You must supply an external definition in one of the compilation units, by designating the inline definition as being external in that compilation unit.

- `#pragma inline` is similar to the `inline` keyword, but with the difference that the compiler always uses C++ inline semantics.

By using the `#pragma inline` directive you can also disable the compiler's heuristics to either force inlining or completely disable inlining. For more information, see *inline*, page 257.

- `--use_cplusplus_inline` forces the compiler to use C++ semantics when compiling a Standard C source code file.
- `--no_inline`, `#pragma optimize=no_inline`, and `#pragma inline=never` all disable function inlining. By default, function inlining is enabled at optimization level High.

The compiler can only inline a function if the definition is known. Normally, this is restricted to the current translation unit. However, when the `--mfc` compiler option for multi-file compilation is used, the compiler can inline definitions from all translation units in the multi-file compilation unit. For more information, see *Multi-file compilation units*, page 177.

For more information about the function inlining optimization, see *Function inlining*, page 180.



# Placing code and data

This chapter describes how the linker handles memory and introduces the concept of segments. It also describes how they correspond to the memory and function types, and how they interact with the runtime environment. The methods for placing segments in memory, which means customizing a linker configuration file, are described.

The intended readers of this chapter are the system designers that are responsible for mapping the segments of the application to appropriate memory areas of the hardware system.

---

## Segments and memory

In an embedded system, there might be many different types of physical memory. Also, it is often critical *where* parts of your code and data are located in the physical memory. For this reason it is important that the development tools meet these requirements.

### WHAT IS A SEGMENT?

A *segment* is a container for pieces of data or code that should be mapped to a location in physical memory. Each segment consists of one or more *segment parts*. Normally, each function or variable with static storage duration is placed in a separate segment part. A segment part is the smallest linkable unit, which allows the linker to include only those segment parts that are referred to. A segment can be placed either in RAM or in ROM. Segments that are placed in RAM generally do not have any content, they only occupy space.

**Note:** Here, ROM memory means all types of read-only memory including flash memory.

The compiler has several predefined segments for different purposes. Each segment is identified by a name that typically describes the contents of the segment, and has a *segment memory type* that denotes the type of content. In addition to the predefined segments, you can also define your own segments.

At compile time, the compiler assigns code and data to the various segments. The IAR XLINK Linker is responsible for placing the segments in the physical memory range, in accordance with the rules specified in the linker configuration file. Ready-made linker configuration files are provided, but, if necessary, they can be modified according to the requirements of your target system and application. It is important to remember that,

from the linker's point of view, all segments are equal; they are simply named parts of memory.

### Segment memory type

Each segment always has an associated segment memory type. In some cases, an individual segment has the same name as the segment memory type it belongs to, for example `CODE`. Make sure not to confuse the segment name with the segment memory type in those cases.

By default, the compiler uses these XLINK segment memory types:

Segment memory type	Description
CODE	For executable code
CONST	For data placed in ROM
DATA	For data placed in RAM

*Table 5: XLINK segment memory types*

XLINK supports several other segment memory types than the ones described above. However, they exist to support other types of microprocessors.

For more information about individual segments, see the chapter *Segment reference*.

---

## Placing segments in memory

The placement of segments in memory is performed by the IAR XLINK Linker. It uses a linker configuration file that contains command line options which specify the locations where the segments can be placed, thereby assuring that your application fits on the target chip. To use the same source code with different devices, just rebuild the code with the appropriate linker configuration file.

In particular, the linker configuration file specifies:

- The placement of segments in memory
- The maximum stack size
- The maximum heap size.

This section describes the most common linker commands and how to customize the linker configuration file to suit the memory layout of your target system. For showing the methods, fictitious examples are used.

### CUSTOMIZING THE LINKER CONFIGURATION FILE

The `config` directory contains ready-made linker configuration files for all supported devices (filename extension `.xcl`). The files contain the information required by the

linker, and are ready to be used. The only change you will normally have to make to the supplied linker configuration file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you must add details about the external RAM memory area.

As an example, we can assume that the target system has this memory layout:

Range	Type
0x0000–0x1FFF	RAM
0x2000–0xCFFF	ROM
0x10000–0x11FFF	RAM
0x20000–0x3FFFF	ROM

*Table 6: Memory layout of a target system (example)*

The ROM can be used for storing `CONST` and `CODE` segment memory types. The RAM memory can contain segments of `DATA` type. The main purpose of customizing the linker configuration file is to verify that your application code and data do not cross the memory range boundaries, which would lead to application failure.

Do not modify the original file. We recommend that you make a copy in the working directory, and modify and use the copy instead.

### The contents of the linker configuration file

Among other things, the linker configuration file contains three different types of XLINK command line options:

- The CPU used:  
`-ccr16c`  
 This specifies your target microprocessor.
- Definitions of constants used in the file. These are defined using the XLINK option `-D`.
- The placement directives (the largest part of the linker configuration file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

In the linker configuration file, numbers are generally specified in hexadecimal format. However, neither the prefix `0x` nor the suffix `h` is necessarily used.

**Note:** The supplied linker configuration file includes comments explaining the contents. See the *IAR Linker and Library Tools Reference Guide* for more information.

## Using the **-Z** command for sequential placement

Use the **-Z** command when you must keep a segment in one consecutive chunk, when you must preserve the order of segment parts in a segment, or, more unlikely, when you must put segments in a specific order.

The following illustrates how to use the **-z** command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x2000-0xCFFF`.

```
-Z (CONST) MYSEGMENTA, MYSEGMENTB=2000-CFFF
```

To place two segments of different types consecutively in the same memory area, do not specify a range for the second segment. In the following example, the `MYSEGMENTA` segment is first located in memory. Then, the rest of the memory range could be used by `MYCODE`.

```
-Z (CONST) MYSEGMENTA=2000-CFFF
-Z (CODE) MYCODE
```

Two memory ranges can partially overlap. This allows segments with different placement requirements to share parts of the memory space; for example:

```
-Z (CONST) MYSMALLSEGMENT=2000-20FF
-Z (CONST) MYLARGESEGMENT=2000-CFFF
```

Even though it is not strictly required, make sure to always specify the end of each memory range. If you do this, the IAR XLINK Linker will alert you if your segments do not fit in the available memory.

## Using the **-P** command for packed placement

The **-P** command differs from **-z** in that it does not necessarily place the segments (or segment parts) sequentially. With **-P** it is possible to put segment parts into holes left by earlier placements.

The following example illustrates how the XLINK **-P** option can be used for making efficient use of the memory area. This command will place the data segment `MYDATA` in `DATA` memory (that is, in RAM) in a fictitious memory range:

```
-P (DATA) MYDATA=0-1FFF, 10000-11FFF
```

If your application has an additional RAM area in the memory range `0xF000-0xF7FF`, you can simply add that to the original definition:

```
-P (DATA) MYDATA=0-1FFF, F000-F7FF, 10000-11FFF
```

The linker can then place some parts of the `MYDATA` segment in the first range, and some parts in the second range. If you had used the **-z** command instead, the linker would have to place all segment parts in the same range.

**Note:** Copy initialization segments—`BASENAME_I` and `BASENAME_ID`—and dynamic initialization segments must be placed using `-Z`.

---

## Data segments

This section contains descriptions of the segments used for storing the different types of data: static, stack, heap, and located.

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different data models available in the compiler. For information about these details, see the chapter *Data storage*.

### STATIC MEMORY SEGMENTS

Static memory is memory that contains variables that are global or declared static, see the chapter *Data storage*. Variables declared static can be divided into these categories:

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero
- Variables that are located by use of the `@` operator or the `#pragma location` directive
- Variables that are declared as `const` and therefore can be stored in ROM
- Variables defined with the `__no_init` keyword, meaning that they should not be initialized at all.

For the static memory segments it is important to be familiar with:

- The segment naming
- How the memory types correspond to segment groups and the segments that are part of the segment groups
- Restrictions for segments holding initialized data
- The placement and size limitation of the segments of each group of static memory segments.

### Segment naming

The names of the segments consist of two parts—the segment group name and a *suffix*—for instance, `DATA32_Z`. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for

example `DATA32` and `__data32`. The following table summarizes the memory types and the corresponding segment groups:

Memory type	Segment group	Memory range
Data16	<code>DATA16_suffix</code>	0x0-0xFFFF
Data20	<code>DATA20_suffix</code>	0x0-0xFFFFF
Data24	<code>DATA24_suffix</code>	0x0-0xFFFFF
Data32	<code>DATA32_suffix</code>	0x0-0xFFFFFFFF
Index4	<code>INDEX4_suffix</code>	<i>base to base + 14 bytes*</i>
Index5	<code>INDEX5_suffix</code>	<i>base to base + 28 bytes*</i>
Index20	<code>INDEX20_suffix</code>	<i>base to base + 0xFFFFF*</i>

Table 7: Memory types with corresponding segment groups

\* *base* is the same for `index4`, `index5`, and `index20`.

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more information about segment memory types, see *Segment memory type*, page 66.

This table summarizes the different suffixes, which XLINK segment memory type they are, and which category of declared data they denote:

Categories of declared data	Suffix	Segment memory type
Non-initialized data	N	DATA
Zero-initialized data	Z	DATA
Non-zero initialized data	I	DATA
Initializers for the above	ID	CONST
Constants	C	CONST

Table 8: Segment name suffixes

For information about all supported segments, see *Summary of segments*, page 293.

### Examples

These examples demonstrate how declared data is assigned to specific segments:

```
__data20 int j;
__data20 int i = 0;
```

The `data20` variables that are to be initialized to zero when the system starts are placed in the segment `DATA20_Z`.

`__no_init __data20 int j;` The data20 non-initialized variables are placed in the segment `DATA20_N`.

`__data20 int j = 4;` The data20 non-zero initialized variables are placed in the segment `DATA20_I` in RAM, and the corresponding initializer data in the segment `DATA20_ID` in ROM.

### Initialized data

When an application is started, the system startup code initializes static and global variables in these steps:

- 1 It clears the memory of the variables that should be initialized to zero.
- 2 It initializes the non-zero variables by copying a block of ROM to the location of the variables in RAM. This means that the data in the ROM segment with the suffix `ID` is copied to the corresponding `I` segment.

This works when both segments are placed in continuous memory. However, if one of the segments is divided into smaller pieces, it is important that:

- The other segment is divided in exactly the same way
- It is legal to read and write the memory that represents the gaps in the sequence.

For example, if the segments are assigned these ranges, the copy might fail:

```
DATA16_I           0x1000-0x10FF and 0x1200-0x12FF
DATA16_ID          0x4000-0x41FF
```

However, in the following example, the linker will place the content of the segments in identical order, which means that the copy will work appropriately:

```
DATA16_I           0x1000-0x10FF and 0x1200-0x12FF
DATA16_ID          0x4000-0x40FF and 0x4200-0x42FF
```

The `ID` segment can, for all segment groups, be placed anywhere in memory, because it is not accessed using the corresponding access method. Note that the gap between the ranges will also be copied.

- 3 Finally, global C++ objects are constructed, if any.

## Data segments for static memory in the default linker configuration file

The default linker configuration file contains these directives to place the static data segments:

```
/* First, the segments to be placed in ROM are defined. */
-Z (CONST) DATA16_C=2000-CFFF
-Z (CONST) DATA20_C=20000-3FFFFFF
-Z (CONST) DATA16_ID, DATA20_ID

/* Then, the RAM data segments are placed in memory. */
-Z (DATA) DATA16_I, DATA16_Z, DATA16_N=0-1FFF
-Z (DATA) DATA20_I, DATA20_Z, DATA20_N=10000-11FFF
```

All the data segments are placed in the area used by on-chip RAM.

## THE STACK

The stack is used by functions to store variables and other information that is used locally by functions, see the chapter *Data storage*. It is a continuous block of memory pointed to by the processor stack pointer register `USP`.

The data segment used for holding the stack is called `CSTACK`. The system startup code initializes the stack pointer to point to the end of the stack segment.

Allocating a memory area for the stack is done differently using the command line interface as compared to when using the IDE.



### Stack size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **Stack/Heap** tab. Add the required stack sizes in the **Stack size** text boxes.



### Stack size allocation from the command line

The size of the `CSTACK` and `ISTACK` segments are defined in the linker configuration file.

The default linker file sets up a constant representing the size of the stacks, at the beginning of the linker file:

```
-D_CSTACK_SIZE=size
-D_ISTACK_SIZE=size
```

**Note:** Normally, this line is prefixed with the comment character `//`. To make the directive take effect, remove the comment character.



Specify an appropriate size for your application. Note that the size is written hexadecimally, but not necessarily with the `0x` notation.



### Placement of stack segment

Further down in the linker file, the actual stack segment is defined in the memory area available for the stack:

```
-Z (DATA) CSTACK+_CSTACK_SIZE=C000-FFFF
-Z (DATA) ISTACK+_ISTACK_SIZE=C000-FFFF
```

**Note:** The range does not specify the size of the stack; it specifies the range of the available memory.



### Stack size considerations

The compiler uses the internal data stack, `CSTACK`, for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too large, RAM is 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 program variables 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.

## THE HEAP

The heap contains dynamic data allocated by 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:

- The linker segments used for the heap
- The steps involved for allocating the heap size, which differs depending on which build interface you are using
- The steps involved for placing the heap segments in memory.

### Heap segments in DLIB

To access a heap in a specific memory, use the appropriate memory attribute as a prefix to the standard functions `malloc`, `free`, `calloc`, and `realloc`, for example:

```
__data16_malloc
```

If you use any of the standard functions without a prefix, the function will use the default heap for the memory type.

Each heap will reside in a segment with the name `_HEAP` prefixed by a memory attribute, for example `DATA16_HEAP`.

For information about available heaps, see *Heaps*, page 109.

## Heap segments in CLIB

CLIB only supports one heap. The memory allocated to the heap can be placed in either data16 memory or in data32 memory, in a segment called either `DATA16_HEAP` or `DATA32_HEAP`. The heap is only included in the application if dynamic memory allocation is actually used.



### Heap size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **Stack/Heap** tab.

Add the required heap sizes in the **Heap size** text boxes.



### Heap size allocation from the command line

The size of the heap segments are defined in the linker configuration file.

The default linker file sets up a constant, representing the size of the heap, at the beginning of the linker file:

```
-D_HEAP16_SIZE=size
-D_HEAP32_SIZE=size
```

Normally, these lines are prefixed with the comment character `//` because the IDE controls the heap size allocation. To make the directive take effect, remove the comment character.

Specify the appropriate size for your application. If you use a heap, you must allocate at least 50 bytes for it. Note that the size is written hexadecimally, but not necessarily with the `0x` notation.



### Placement of heap segment

The actual heap segment is allocated in the memory area available for the heap:

```
-Z (DATA) DATA16_HEAP+_HEAP16_SIZE=8000-8FFF
-Z (DATA) DATA32_HEAP+_HEAP32_SIZE=10000-11FFF
```

**Note:** The range does not specify the size of the heap; it specifies the range of the available memory.



### Heap size and standard I/O

If your DLIB runtime environment is configured to use `FILE` descriptors, as in the Full configuration, input and output buffers for file handling will be allocated. In that case, 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 CR16C microprocessor. 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.

## LOCATED DATA

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in either the `ABSOLUTE_C` or the `ABSOLUTE_N` segment. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker configuration file.

## USER-DEFINED SEGMENTS

If you create your own segments by using for example the `#pragma location` directive or the `@` operator, these segments must also be defined in the linker configuration file using the `-Z` or `-P` segment control directives.

---

## Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For information about all segments, see *Summary of segments*, page 293.

### STARTUP CODE

The segment `CSTART` contains code used during system startup, runtime initialization (`cstartup`), and system termination (`cexit`). The system startup code should be placed at the location where the chip starts executing code after a reset. For the CR16C microprocessor, this is at the address `0x0000`. This address depends on the device you are using. For more information about your device, see the device description file. The segments must also be placed into one continuous memory space, which means that the `-P` segment directive cannot be used.

In the default linker configuration file, this line will place the `CSTART` segment at the address `0x0000`:

```
-Z (CODE) CSTART=0000
```

## NORMAL CODE

Code for normal functions is placed in the `CODE` segment. Again, this is a simple operation in the linker command file:

```
-Z (CODE) CODE=0000-BFFF, C000-1FFFFFF
```

## INTERRUPT VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the segment `INTVEC`. The linker directive in the default linker configuration file places the `INTVEC` segment in the same areas as the `CODE` segment by not specifying a range:

```
-Z (CONST) INTVEC
```

---

## C++ dynamic initialization

In C++, all global objects are created before the `main` function is called. The creation of objects can involve the execution of a constructor.

The `DIFUNCT` segment contains a vector of addresses that point to initialization code. All entries in the vector are called when the system is initialized.

For example:

```
-Z (CONST) DIFUNCT=0000-BFFF
```

`DIFUNCT` must be placed using `-z`. For additional information, see *DIFUNCT*, page 304.

---

## Verifying the linked result of code and data placement

The linker has several features that help you to manage code and data placement, for example, messages at link time and the linker map file.

### SEGMENT TOO LONG ERRORS AND RANGE ERRORS

All code or data that is placed in relocatable segments will have its absolute addresses resolved at link time. Note that it is not known until link time whether all segments will fit in the reserved memory ranges. If the contents of a segment do not fit in the address range defined in the linker configuration file, `XLINK` will issue a *segment too long* error.

Some instructions do not work unless a certain condition holds after linking, for example that a branch must be within a certain distance or that an address must be even. `XLINK` verifies that the conditions hold when the files are linked. If a condition is not

satisfied, XLINK generates a *range error* or warning and prints a description of the error.

For more information about these types of errors, see the *IAR Linker and Library Tools Reference Guide*.

## LINKER MAP FILE

XLINK can produce an extensive cross-reference listing, which can optionally contain the following information:

- A segment map which lists all segments in dump order
- A module map which lists all segments, local symbols, and entries (public symbols) for every module in the program. All symbols not included in the output can also be listed
- A module summary which lists the contribution (in bytes) from each module
- A symbol list which contains every entry (global symbol) in every module.

Use the option **Generate linker listing** in the IDE, or the option `-x` on the command line, and one of their suboptions to generate a linker listing.

Normally, XLINK will not generate an output file if any errors, such as range errors, occur during the linking process. Use the option **Always generate output** in the IDE, or the option `-B` on the command line, to generate an output file even if a non-fatal error was encountered.

For more information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IDE Project Management and Building Guide*.

Verifying the linked result of code and data placement

# The DLIB runtime environment

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

DLIB can be used with both the C and the C++ languages. CLIB, on the other hand, can only be used with the C language. For more information, see the chapter *The CLIB runtime environment*.

---

## Introduction to the runtime environment

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

### RUNTIME ENVIRONMENT FUNCTIONALITY

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

The runtime library is delivered both as prebuilt libraries and (depending on your product package) as source files, and you can find them in the product subdirectories `cr16c\lib` and `cr16c\src\lib`, respectively.

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

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
  - Peripheral unit registers and interrupt definitions in include files
  - Target-specific arithmetic support modules like hardware multipliers or floating-point coprocessors.
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.

- A floating-point environment (fenv) that contains floating-point arithmetics support, see *fenv.h*, page 289.
- Special compiler support, for instance functions for switch handling or integer arithmetics.

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

## SETTING UP THE RUNTIME ENVIRONMENT

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

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

- Choose which library to use—the DLIB or the CLIB library  
Use the compiler option `--clib` or `--dlib`, respectively. For more information about the libraries, see *Library overview*, page 283.
- Choose which runtime library object file to use  
The IDE will automatically choose a runtime library based on your project settings. If you build from the command line, you must specify the object file explicitly. See *Using a prebuilt library*, page 81.
- Choose which predefined runtime library configuration to use—Normal or Full  
You can configure the level of support for certain library functionality, for example, locale, file descriptors, and multibyte characters. If you do not specify anything, a default library configuration file that matches the library object file is automatically used. To specify a library configuration explicitly, use the `--dlib_config` compiler option. See *Library configurations*, page 96.
- Optimize the size of the runtime library  
You can specify the formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 84. You can also specify the size and placement of the stacks and the heaps, see *The stack*, page 72, and *The heap*, page 73, respectively.
- Include debug support for runtime and I/O debugging  
The library offers support for mechanisms like redirecting standard input and output to the C-SPY Terminal I/O window and accessing files on the host computer, see *Application debug support*, page 86.



- Adapt the library for target hardware

The library uses a set of low-level functions for handling accesses to your target system. To make these accesses work, you must implement your own version of these functions. For example, to make `printf` write to an LCD display on your board, you must implement a target-adapted version of the low-level function `__write`, so that it can write characters to the display. To customize such functions, you need a good understanding of the library low-level interface, see *Adapting the library for target hardware*, page 89.

- Override library modules

If you have customized the library functionality, you need to make sure your versions of the library modules are used instead of the default modules. This can be done without rebuilding the entire library, see *Overriding library modules*, page 90.

- Customize system initialization

It is likely that you need to customize the source code for system initialization, for example, your application might need to initialize memory-mapped special function registers, or omit the default initialization of data segments. You do this by customizing the routine `__low_level_init`, which is executed before the data segments are initialized. See *System startup and termination*, page 92 and *Customizing system initialization*, page 95.

- Configure your own library configuration files

In addition to the prebuilt library configurations, you can make your own library configuration, but that requires that you *rebuild* the library. This gives you full control of the runtime environment. See *Building and using a customized library*, page 91.

- Check module consistency

You can use runtime model attributes to ensure that modules are built using compatible settings, see *Checking module consistency*, page 110.

---

## Using a prebuilt library

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

- Processor variant
- Code model—Short or Normal
- Data model—Small, Medium, Large, Huge, or Indexed
- Library configuration—Normal or Full.

## CHOOSING A LIBRARY

The IDE will include the correct library object file and library configuration file based on the options you select. See the *IDE Project Management and Building Guide* for more information.

If you build your application from the command line, make the following settings:

- Specify which library object file to use on the XLINK command line, for instance:  
dlcr16csnln.r45
- If you do not specify a library configuration, the default will be used. However, you can specify the library configuration explicitly for the compiler:  
--dlib\_config C:\...\dlcr16cnln.h

**Note:** All modules in the library have a name that starts with the character ? (question mark).

You can find the library object files and the library configuration files in the subdirectory cr16c\lib\dlib.

These prebuilt runtime libraries are available:

Library	CPU	Code model	Data model	Library configuration
dlcr16cssn.r45	cr16c	Short	Small	Normal
dlcr16csmn.r45	cr16c	Short	Medium	Normal
dlcr16csln.r45	cr16c	Short	Large	Normal
dlcr16cshn.r45	cr16c	Short	Huge	Normal
dlcr16cnsn.r45	cr16c	Normal	Small	Normal
dlcr16cnmn.r45	cr16c	Normal	Medium	Normal
dlcr16cnln.r45	cr16c	Normal	Large	Normal
dlcr16cnhn.r45	cr16c	Normal	Huge	Normal
dlcr16cnin.r45	cr16c	Normal	Indexed	Normal
dlcr16cssf.r45	cr16c	Short	Small	Full
dlcr16csmf.r45	cr16c	Short	Medium	Full
dlcr16cslf.r45	cr16c	Short	Large	Full
dlcr16cshf.r45	cr16c	Short	Huge	Full
dlcr16cnsf.r45	cr16c	Normal	Small	Full
dlcr16cnmf.r45	cr16c	Normal	Medium	Full
dlcr16cnlf.r45	cr16c	Normal	Large	Full

Table 9: Prebuilt libraries

Library	CPU	Code model	Data model	Library configuration
dlcr16cnhf.r45	cr16c	Normal	Huge	Full
dlcr16cnif.r45	cr16c	Normal	Indexed	Full

Table 9: Prebuilt libraries (Continued)

## LIBRARY FILENAME SYNTAX

The names of the libraries are constructed from these elements:

{ <i>library</i> }	is dl for the IAR DLIB runtime environment
{ <i>cpu</i> }	is cr16c
{ <i>code_model</i> }	is one of s or n for Short and Normal code model, respectively
{ <i>data_model</i> }	is one of s, m, l, h, or i for the Small, Medium, Large, Huge, and Indexed data model, respectively
{ <i>lib_config</i> }	is one of n or f for Normal and Full, respectively.

**Note:** The library configuration file has the same base name as the library.

## CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

These items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	<i>Choosing formatters for printf and scanf</i> , page 84
Startup and termination code	<i>System startup and termination</i> , page 92
Low-level input and output	<i>Standard streams for input and output</i> , page 97
File input and output	<i>File input and output</i> , page 102
Low-level environment functions	<i>Environment interaction</i> , page 105
Low-level signal functions	<i>Signal and raise</i> , page 106
Low-level time functions	<i>Time</i> , page 106
Some library math functions	<i>Math functions</i> , page 107
Size of heaps, stacks, and segments	<i>Placing code and data</i> , page 65

Table 10: Customizable items

For information about how to override library modules, see *Overriding library modules*, page 90.

## Choosing formatters for printf and scanf

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

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

**Note:** If you rebuild the library, you can optimize these functions even further, see *Configuration symbols for printf and scanf*, page 100.

### CHOOSING A PRINTF FORMATTER

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

This table summarizes the capabilities of the different formatters:

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

Table 11: Formatters for printf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 100.



## Manually specifying the print formatter in the IDE

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



## Manually specifying the printf formatter from the command line

To specify a formatter manually, add one of these lines in the linker configuration file you are using:

```
-e_printfFull=_Printf
-e_printfFullNoMb=_Printf
-e_printfLarge=_Printf
-e_printfLargeNoMb=_Printf
_e_printfSmall=_Printf
-e_printfSmallNoMb=_Printf
-e_printfTiny=_Printf
-e_printfTinyNoMb=_Printf
```

## CHOOSING A SCANF FORMATTER

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

This table summarizes the capabilities of the different formatters:

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

Table 12: Formatters for `scanf`

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for `printf` and `scanf`*, page 100.



### Manually specifying the scanf formatter in the IDE

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



### Manually specifying the scanf formatter from the command line

To specify a formatter manually, add one of these lines in the linker configuration file you are using:

```
-e_ScanfFull=_Scanf
-e_ScanfFullNoMb=_Scanf
-e_ScanfLarge=_Scanf
-e_ScanfLargeNoMb=_Scanf
_e_ScanfSmall=_Scanf
_e_ScanfSmallNoMb=_Scanf
```

---

## Application debug support

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

### INCLUDING C-SPY DEBUGGING SUPPORT

You can make the library provide different levels of debugging support—basic, runtime, and I/O debugging.

This table describes the different levels of debugging support:

Debugging support	Linker option in the IDE	Linker command line option	Description
Basic debugging	<b>Debug information for C-SPY</b>	<code>-Fubrof</code>	Debug support for C-SPY without any runtime support
Runtime debugging*	<b>With runtime control modules</b>	<code>-r</code>	The same as <code>-Fubrof</code> , but also includes debugger support for handling program abort, exit, and assertions.

*Table 13: Levels of debugging support in runtime libraries*

Debugging support	Linker option in the IDE	Linker command line option	Description
I/O debugging*	<b>With I/O emulation modules</b>	<code>-rt</code>	The same as <code>-r</code> , but also includes debugger support for I/O handling, which means that <code>stdin</code> and <code>stdout</code> are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.

Table 13: Levels of debugging support in runtime libraries (Continued)

\* If you build your application project with this level of debugging support, certain functions in the library are replaced by functions that communicate with C-SPY. For more information, see *The debug library functionality*, page 87.

In the IDE, choose **Project>Options>Linker**. On the **Output** page, select the appropriate **Format** option.

On the command line, use any of the linker options `-r` or `-rt`.

## THE DEBUG LIBRARY FUNCTIONALITY

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

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

The mechanism used for implementing this feature works as follows:

The debugger will detect the presence of the function `__DebugBreak`, which will be part of the application if you linked it with the `XLINK` option for C-SPY debugging support. In this case, the debugger will automatically set a breakpoint at the `__DebugBreak` function. When the application calls, for example, `open`, the `__DebugBreak` function is called, which will cause the application to break and perform the necessary services. The execution will then resume.

## THE C-SPY TERMINAL I/O WINDOW

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

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

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

### 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:

```
-e__write_buffered=__write
```



## LOW-LEVEL FUNCTIONS IN THE DEBUG LIBRARY

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

Function in DLIB low-level interface	Response by C-SPY
<code>abort</code>	Notifies that the application has called <code>abort</code> *
<code>clock</code>	Returns the clock on the host computer
<code>__close</code>	Closes the associated host file on the host computer
<code>__exit</code>	Notifies that the end of the application was reached *
<code>__lseek</code>	Searches in the associated host file on the host computer
<code>__open</code>	Opens a file on the host computer
<code>__read</code>	Directs <code>stdin</code> , <code>stdout</code> , and <code>stderr</code> to the Terminal I/O window. All other files will read the associated host file
<code>remove</code>	Writes a message to the Debug Log window and returns <code>-1</code>
<code>rename</code>	Writes a message to the Debug Log window and returns <code>-1</code>
<code>_ReportAssert</code>	Handles failed asserts *
<code>system</code>	Writes a message to the Debug Log window and returns <code>-1</code>
<code>time</code>	Returns the time on the host computer
<code>__write</code>	Directs <code>stdin</code> , <code>stdout</code> , and <code>stderr</code> to the Terminal I/O window. All other files will write to the associated host file

Table 14: Functions with special meanings when linked with debug library

\* The linker option `With I/O emulation modules` is not required for these functions.

**Note:** You should not use the low-level interface functions prefixed with `_` or `__` directly in your application. Instead you should use the high-level functions that use these functions. For more information, see *Library low-level interface*, page 90.

## Adapting the library for target hardware

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

When you have implemented your low-level interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules*, page 90.

## LIBRARY LOW-LEVEL INTERFACE

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

However, the library contains a debug version of the library low-level interface, where the low-level functions are implemented so that they interact with the host computer via the debugger, instead of with the target hardware. If you use the debug library, your application can perform tasks like writing to the Terminal I/O window, accessing files on the host computer, getting the time from the host computer, etc. For more information, see *The debug library functionality*, page 87.

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

The library files that you can override with your own versions are located in the `cr16c\src\lib` directory.

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

- *Standard streams for input and output*, page 97
- *File input and output*, page 102
- *Signal and raise*, page 106
- *Time*, page 106
- *Assert*, page 109.

---

## Overriding library modules

To use a library low-level interface that you have implemented, add it to your application. See *Adapting the library for target hardware*, page 89. Or, you might want to override a default library routine with your customized version. In both cases, follow this procedure:

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

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

The library files that you can override with your own versions are located in the `cr16c\src\lib` directory.

---

## Building and using a customized library

Building a customized library is a complex process. Therefore, consider carefully whether it is really necessary. You must build your own 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, etc.

In those cases, you must:

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

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

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

### SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template uses the Full library configuration, see Table 15, *Library configurations*.



In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 37.

**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. Your library also has its own library configuration file `libraryname.h`, which sets up that specific library with the required library configuration. For more information, see Table 10, *Customizable items*.

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

### Modifying the library configuration file

In your library project, open the file `libraryname.h` 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 **Library file** text box, locate your library file.
- 4 In the **Configuration file** text box, locate your library configuration file.

---

## System startup and termination

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

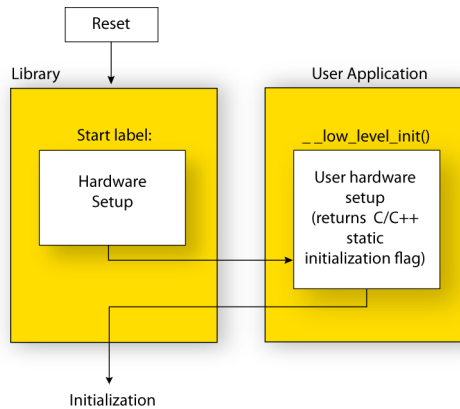
The code for handling startup and termination is located in the source files `cstartup.s45` and `cexit.s45` located in the `cr16c\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 95.

## SYSTEM STARTUP

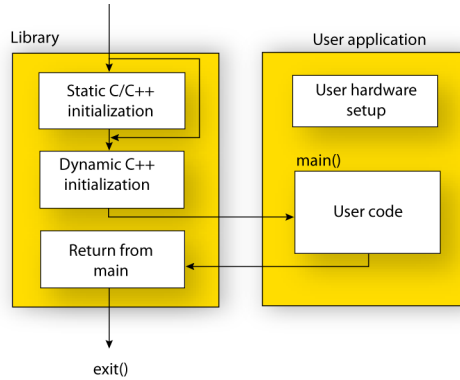
During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

For the hardware initialization, it looks like this:



- The `INTBASE` processor register is initialized to point to the interrupt vector table.
- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code.
- The stack pointer is initialized to the end of the `CSTACK` segment
- The `CFG` processor register is initialized. The `ED` flag is set and the `SR` flag is assigned in correspondence with the code model.
- If indexed addressing is enabled, the corresponding base register—`R13`—is initialized.
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

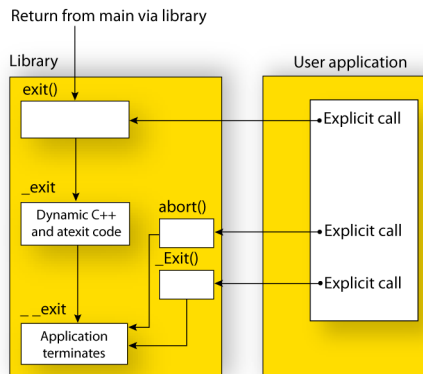
For the C/C++ initialization, it looks like this:



- 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 information, see *Initialized data*, page 71
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

## 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.

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

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

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

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

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

### C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, 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 *Application debug support*, page 86.

---

## 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 segments performed by `cstartup`.

You can do this by writing a customized file with your own version of the routine `__low_level_init`, to be called from `cstartup` before the data segments are initialized. Modifying the file `cstartup.s45` directly should be avoided.

The code for handling system startup, including a basic version of the `__low_level_init` routine, is located in the source file `cstartup.s45`, located in the `cr16c\src\lib` directory.

**Note:** Normally, you do not need to customize either of the files `cstartup.s45` or `cexit.s45`.

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 91.

**Note:** Even if you modify the file `cstartup.s45`, you do not have to rebuild the library.

## **\_\_LOW\_LEVEL\_INIT**

The routine `low_level_init` can be implemented using C or assembler. The only limitation using a 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 segments should be initialized by the system startup code. If the function returns 0, the data segments will not be initialized.

**Note:** The file `intrinsics.h` must be included by a file that defines `low_level_init` to assure correct behavior of the `__low_level_init` routine.

## **MODIFYING THE FILE CSTARTUP.S45**

As noted earlier, you should not modify the file `cstartup.s45` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s45`, 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 90.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s45`. For information about how to change the start label used by the linker, read about the `-s` option in the *IAR Linker and Library Tools Reference Guide*.

---

## **Library configurations**

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

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

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



These predefined library configurations are available:

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

Table 15: Library configurations

## CHOOSING A RUNTIME CONFIGURATION

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

- Default prebuilt configuration—if you do not specify a library configuration explicitly you will get the default configuration. A configuration file that matches the runtime library object file will automatically be used.
- Prebuilt configuration of your choice—to specify a runtime configuration explicitly, use the `--dlib_config` compiler option. See `--dlib_config`, page 207.
- Your own configuration—you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For more information, see *Building and using a customized library*, page 91.

The prebuilt libraries are based on the default configurations, see Table 15, *Library configurations*.

---

## Standard streams for input and output

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

There are low-level I/O functions, which are the fundamental functions through which C and C++ performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these functions using whatever facilities the hardware environment provides. For more information about implementing low-level functions, see *Adapting the library for target hardware*, page 89.

## 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 `cr16c\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 91. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

**Note:** If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *Application debug support*, page 86.

### Example of using `__write`

The code in this example uses memory-mapped I/O to write to an LCD display, whose port is assumed to be located at address `0xFF4942`:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 0xFF4942;

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

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

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

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

    return nChars;
}
```

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

### Example of using `__read`

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at 0xFF4942:

```
#include <stddef.h>
__no_init volatile unsigned char kbIO @ 0xFF4942;
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 173.

---

## 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 84.

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

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

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

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

Table 16: Descriptions of `printf` configuration symbols

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

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

Table 17: Descriptions of `scanf` configuration symbols

## CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must:

- 1 Set up a library project, see *Building and using a customized library*, page 91.
- 2 Define the configuration symbols according to your application requirements.

---

## File input and output

The library contains a large number of powerful functions for file I/O operations, such as `fopen`, `fclose`, `fprintf`, `fputs`, etc. All these functions call a small set of low-level functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters. Before your application can use the library functions for file I/O operations, you must implement the corresponding low-level function to suit your target hardware. For more information, see *Adapting the library for target hardware*, page 89.

Note that file I/O capability in the library is only supported by libraries with the full library configuration, see *Library configurations*, page 96. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

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

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

Table 18: Low-level I/O files

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

**Note:** If you link your application with I/O debug support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Application debug support*, page 86.

---

## Locale

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

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

The DLIB library can be used in two main modes:

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

## LOCALE SUPPORT IN PREBUILT LIBRARIES

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

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

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

## CUSTOMIZING THE LOCALE SUPPORT

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

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

### Locale configuration symbols

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

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

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 91.

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

### Building a library without support for locale interface

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

### Building a library with support for locale interface

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

## CHANGING LOCALES AT RUNTIME

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

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

*lang\_REGION*

or

*lang\_REGION.encoding*

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

The *lang\_REGION* part matches the `_LOCALE_USE_LANG_REGION` preprocessor symbols that can be specified in the library configuration file.



## Example

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

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

---

## Environment interaction

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

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

### THE GETENV FUNCTION

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `cr16c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 90.

### THE SYSTEM FUNCTION

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

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 91.

**Note:** If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For more information, see *Application debug support*, page 86.

---

## 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 `cr16c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 90.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 91.

---

## Time

To make the `__time32`, and `date` functions work, you must implement the functions `clock`, `__time32`, and `__getzone`.

To implement these functions does not require that you rebuild the library. You can find source templates in the files `clock.c`, `time.c`, and `getzone.c` in the `cr16c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 90.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 91.

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

**Note:** If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For more information, see *Application debug support*, page 86.

---

## Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make `strtod` accept hexadecimal floating-point strings, you must:

- 1 Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.
- 2 Rebuild the library, see *Building and using a customized library*, page 91.

---

## Math functions

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

### SMALLER VERSIONS

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

The names of the functions are constructed like:

```
__iar_XXX_small<f|l>
```

where `f` is used for `float` variants, `l` is used for `long double` variants, and no suffix is used for `double` variants.

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

```
-e __iar_sin_small=sin
-e __iar_cos_small=cos
-e __iar_tan_small=tan
-e __iar_log_small=log
-e __iar_log10_small=log10
-e __iar_exp_small=exp
-e __iar_pow_small=pow
-e __iar_Sin_small=__iar_Sin
```

```
-e __iar_sin_smallf=sinf
-e __iar_cos_smallf=cosf
-e __iar_tan_smallf=tanf
-e __iar_log_smallf=logf
-e __iar_log10_smallf=log10f
-e __iar_exp_smallf=expf
-e __iar_pow_smallf=powf
-e __iar_Sin_smallf=__iar_Sinf
```

```
-e __iar_sin_smalll=sinl
-e __iar_cos_smalll=cosl
-e __iar_tan_smalll=tanl
-e __iar_log_smalll=logl
-e __iar_log10_smalll=log10l
-e __iar_exp_smalll=expl
-e __iar_pow_smalll=powl
-e __iar_Sin_smalll=__iar_Sinl
```

Note that if `cos` or `sin` is redirected, `__iar_Sin` must be redirected as well.

## MORE ACCURATE VERSIONS

The functions `cos`, `pow`, `sin`, and `tan`, and the help functions `__iar_Sin` and `__iar_Pow` exist in versions in the library that are more exact and can handle larger argument ranges. The drawback is that they are larger and slower than the default versions.

The names of the functions are constructed like:

```
__iar_xxx_accurate<f|l>
```

where `f` is used for `float` variants, `l` is used for `long double` variants, and no suffix is used for `double` variants.

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

```
-e __iar_sin_accurate=sin
-e __iar_cos_accurate=cos
-e __iar_tan_accurate=tan
-e __iar_pow_accurate=pow
-e __iar_Sin_accurate=__iar_Sin
-e __iar_Pow_accurate=__iar_Pow
```

```

-e__iar_sin_accuratef=sinf
-e__iar_cos_accuratef=cosf
-e__iar_tan_accuratef=tanf
-e__iar_pow_accuratef=powf
-e__iar_Sin_accuratef=__iar_Sinf
-e__iar_Pow_accuratef=__iar_Powf

-e__iar_sin_accuratel=sinl
-e__iar_cos_accuratel=cosl
-e__iar_tan_accuratel=tanl
-e__iar_pow_accuratel=powl
-e__iar_Sin_accuratel=__iar_Sinl
-e__iar_Pow_accuratel=__iar_Powl

```

Note that if `cos` or `sin` is redirected, `__iar_Sin` must be redirected as well. The same applies to `pow` and `__iar_Pow`.

---

## Assert

If you linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `cr16c\src\lib` directory. For more information, see *Building and using a customized library*, page 91. 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 282.

---

## Heaps

The runtime environment supports heaps in these memory types:

Memory type	Segment name	Memory attribute	Used by default in data model
Data16	DATA16_HEAP	__data16	Small
Data32	DATA32_HEAP	__data32	Medium, Large, Huge, Indexed

Table 19: Heaps and memory types

See *The heap*, page 73 for information about how to set the size for each heap. To use a specific heap, the prefix in the table is the memory attribute to use in front of `malloc`, `free`, `calloc`, and `realloc`, for example `__data16_malloc`. The default functions

will use one of the specific heap variants, depending on project settings such as data model. For information about how to use a specific heap in C++, see *New and Delete operators*, page 159.

---

## Checking module consistency

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

For example, in the compiler, it is possible to specify the code model. If you write a routine that only works for the normal code model, it is possible to check that the routine is not used in an application built using the short code model.

### RUNTIME MODEL ATTRIBUTES

A runtime attribute is a pair constituted of a named key and its corresponding value. 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.

### Example

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

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

*Table 20: Example of runtime model attributes*

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

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

## USING RUNTIME MODEL ATTRIBUTES

To ensure module consistency with other object files, use the `#pragma rtmodel` directive to specify runtime model attributes in your C/C++ source code. For example, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example `uart`. For each mode, specify a value, for example `mode1` and `mode2`. Declare this in each module that assumes that the UART is in a particular mode. This is how it could look like in one of the modules:

```
#pragma rtmodel="uart", "mode1"
```

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

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

Note that key names that start with two underscores are reserved by the compiler. For more information about the syntax, see *rtmodel*, page 263 and the *IAR Assembler Reference Guide for CR16C*, respectively.

**Note:** The predefined runtime attributes all start with two underscores. Any attribute names you specify yourself should not contain two initial underscores in the name, to eliminate any risk that they will conflict with future IAR runtime attribute names.

At link time, the IAR XLINK 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.

## PREDEFINED RUNTIME ATTRIBUTES

The table below shows the predefined runtime model attributes that are available for the compiler. These can be included in assembler code or in mixed C/C++ and assembler code.

Runtime model attribute	Value	Description
<code>__rt_version</code>	3	This runtime key is always present in all modules generated by the compiler. If a major change in the runtime characteristics occurs, the value of this key changes.
<code>__code_model</code>	short or normal	Corresponds to the code model used in the project.

Table 21: Predefined runtime model attributes

Runtime model attribute	Value	Description
<code>__data_model</code>	small, medium, large, huge, or indexed	Corresponds to the data model used in the project.
<code>__reg_r13</code>	free, index_base, or undefined	Corresponds to the use of register R13, or undefined when R13 is not used.

Table 21: Predefined runtime model attributes (Continued)

The easiest way to find the proper settings of the `RTMODEL` directive is to compile a C or C++ module to generate an assembler file, and then examine the file.

If you are using assembler routines in the C or C++ code, see the chapter *Assembler directives* in the *IAR Assembler Reference Guide for CR16C*.

## Examples

The following assembler source code provides a function that increases the register R13 to count the number of times it was called. The routine assumes that the application does not use R13 for anything else, that is, the register is locked for usage. To ensure this, a runtime module attribute, `__reg_r13`, is defined with a value `counter`. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not set this attribute.

```

        module          myCounter
        public         myCounter
        section        CODE:CODE
        rtmodel        "__reg_r13", "counter"

myCounter: addw        r2, r13
           movw        r13, r0
           jump        (ra)
           end

```

If this module is used in an application that contains modules where the register R4 is not locked, the linker issues an error:

```

Error[e117]: Incompatible runtime models. Module myCounter
specifies that '__reg_r13' must be 'counter', but module part1
has the value 'free'

```



# The CLIB runtime environment

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

The standard library uses a small set of low-level input and output routines for character-based I/O. This chapter describes how the low-level routines can be replaced by your own version. The chapter also describes how you can choose printf and scanf formatters.

The chapter then describes system initialization and termination. It presents how an application can control what happens before the start function main is called, and the method for how you can customize the initialization. Finally, the C-SPY® runtime interface is covered.

Note that CLIB does not support any C99 functionality. For example, complex numbers and variable length arrays are not supported.

Note that the legacy CLIB runtime environment is provided for backward compatibility and should not be used for new application projects.

For information about migrating from CLIB to DLIB, see the *IAR Embedded Workbench® Migration Guide*.

---

## Using a prebuilt library

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

- Code model—Short or Normal
- Data model—Small, Medium, Large, Huge, or Indexed

## CHOOSING A LIBRARY

The IDE includes the correct runtime library based on the options you select. See the *IDE Project Management and Building Guide* for more information.

Specify which runtime library object file to use on the XLINK command line, for instance:

```
clcr16csl.r45
```

The CLIB runtime environment includes the C standard library. The linker will include only those routines that are required—directly or indirectly—by your application. For more information about the runtime libraries, see the chapter *Library functions*.

These prebuilt libraries are available:

Library object file	CPU	Code model	Data model
clcr16css.r45	cr16c	Short	Small
clcr16csm.r45	cr16c	Short	Medium
clcr16csl.r45	cr16c	Short	Large
clcr16csh.r45	cr16c	Short	Huge
clcr16cns.r45	cr16c	Normal	Small
clcr16cnm.r45	cr16c	Normal	Medium
clcr16cnl.r45	cr16c	Normal	Large
clcr16cnh.r45	cr16c	Normal	Huge
clcr16cni.r45	cr16c	Normal	Indexed

Table 22: Runtime libraries

### Library filename syntax

The runtime library names are constructed in this way:

```
{type}{cpu}{code_model}{data_model}.r45
```

where

- `{type}` is `cl` for the IAR CLIB L library
- `{cpu}` is `cr16c`
- `{code_model}` is one of `s` or `n` for Short and Normal code model, respectively
- `{data_model}` is one of `s`, `m`, `l`, `h`, or `i` for the Small, Medium, Large, Huge, and Indexed data model, respectively.

## Input and output

You can customize:

- The functions related to character-based I/O
- The formatters used by `printf/sprintf` and `scanf/sscanf`.

### CHARACTER-BASED I/O

The functions `putchar` and `getchar` are the fundamental C functions for character-based I/O. For any character-based I/O to be available, you must provide definitions for these two functions, using whatever facilities the hardware environment provides.

The creation of new I/O routines is based on these files:

- `putchar.c`, which serves as the low-level part of functions such as `printf`
- `getchar.c`, which serves as the low-level part of functions such as `scanf`.

The code example below shows how memory-mapped I/O could be used to write to a memory-mapped I/O device:

```
__no_init volatile unsigned char devIO @ 0xFF4942;

int putchar(int outChar)
{
    devIO = outChar;
    return outChar;
}
```

The exact address is a design decision. For example, it can depend on the selected processor variant.

For information about how to include your own modified version of `putchar` and `getchar` in your project build process, see *Overriding library modules*, page 90.

### FORMATTERS USED BY PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter, called `_formatted_write`. There are three variants of the formatter:

```
_large_write
_medium_write
_small_write
```

By default, the linker automatically uses the most appropriate formatter for your application.

### **`_large_write`**

The `_large_write` formatter supports the C89 `printf` format directives.

### **`_medium_write`**

The `_medium_write` formatter has the same format directives as `_large_write`, except that floating-point numbers are not supported. Any attempt to use a `%f`, `%g`, `%G`, `%e`, or `%E` specifier will produce a runtime error:

```
FLOATS? wrong formatter installed!
```

`_medium_write` is considerably smaller than the large version.

### **`_small_write`**

The `_small_write` formatter works in the same way as `_medium_write`, except that it supports only the `%%`, `%d`, `%o`, `%c`, `%s`, and `%x` specifiers for integer objects, and does not support field width or precision arguments. The size of `_small_write` is 10–15% that of `_medium_write`.



### **Specifying the printf formatter in the IDE**

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library Options** tab.
- 2 Choose the appropriate **Printf formatter** option, which can be either **Auto**, **Small**, **Medium**, or **Large**.



### **Specifying the printf formatter from the command line**

To explicitly specify and override the formatter used by default, add one of the following lines to the linker configuration file:

```
-e_small_write=_formatted_write
-e_medium_write=_formatted_write
-e_large_write=_formatted_write
```

### **Customizing printf**

For many embedded applications, `sprintf` is not required, and even `printf` with `_small_write` provides more facilities than are justified, considering the amount of memory it consumes. Alternatively, a custom output routine might be required to support particular formatting needs or non-standard output devices.

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 90.

## FORMATTERS USED BY SCANF AND SSCANF

As with the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter, called `_formatted_read`. There are two variants of the formatter:

```
_large_read
_medium_read
```

By default, the linker automatically uses the most appropriate formatter for your application.

### **`_large_read`**

The `_large_read` formatter supports the C89 `scanf` format directives.

### **`_medium_read`**

The `_medium_read` formatter has the same format directives as the large version, except that floating-point numbers are not supported. `_medium_read` is considerably smaller than the large version.



### **Specifying the `scanf` formatter in the IDE**

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library Options** tab.
- 2 Choose the appropriate **Scanf formatter** option, which can be either **Auto**, **Medium** or **Large**.



### **Specifying the read formatter from the command line**

To explicitly specify and override the formatter used by default, add one of the following lines to the linker configuration file:

```
-e_medium_read=_formatted_read
-e_large_read=_formatted_read
```

---

## System startup and termination

This section describes the actions the runtime environment performs during startup and termination of applications.

The code for handling startup and termination is located in the source files `cstartup.s45` and `cexit.s45`, located in the `cr16c\src\lib` directory.

**Note:** Normally, you do not need to customize either of the files `cstartup.s45` or `cexit.s45`.

## SYSTEM STARTUP

When an application is initialized, several steps are performed:

- The `INTBASE` processor register is initialized to point to the interrupt vector table.
- The custom function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations
- The stack pointer is initialized to the end of the `CSTACK` segment
- The `CFG` processor register is initialized. The `ED` flag is set and the `SR` flag is assigned in correspondence with the code model.
- If indexed addressing is enabled, the corresponding base register—`R13`—is initialized.
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables
- The `main` function is called, which starts the application.

Note that the system startup code contains code for more steps than described here. The other steps are applicable to the `DLIB` runtime environment.

## SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

Because the C standard states that the two methods should be equivalent, the `cstartup` 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 assembler.

When the application is built in debug mode, `C-SPY` stops when it reaches the special code label `?C_EXIT`.

An application can also exit by calling the `abort` function. The default function just calls `__exit` to halt the system, without performing any type of cleanup.

---

## Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 90, in the chapter *The DLIB runtime environment*.

---

## Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 95.

---

## C-SPY runtime interface

The low-level debugger interface is used for communication between the application being debugged and the debugger itself. The interface is simple: C-SPY will place breakpoints on certain assembler labels in the application. When code located at the special labels is about to be executed, C-SPY will be notified and can perform an action.

### THE DEBUGGER TERMINAL I/O WINDOW

When code at the labels `?C_PUTCHAR` and `?C_GETCHAR` is executed, data will be sent to or read from the debugger window.

For the `?C_PUTCHAR` routine, one character is taken from the output stream and written. If everything goes well, the character itself is returned, otherwise `-1` is returned.

When the label `?C_GETCHAR` is reached, C-SPY returns the next character in the input field. If no input is given, C-SPY waits until the user types some input and presses the Return key.

To make the Terminal I/O window available, the application must be linked with the XLINK option **With I/O emulation modules** selected. See the *IDE Project Management and Building Guide*.

### TERMINATION

The debugger stops executing when it reaches the special label `?C_EXIT`.

---

## Checking module consistency

For information about how to check module consistency, see *Checking module consistency*, page 110.





# 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 CR16C microprocessor 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 in the different code models, the different memory access methods corresponding to the supported memory types, 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 CR16C provides several ways to access low-level resources:

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

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

### INTRINSIC FUNCTIONS

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

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

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

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

## MIXING C AND ASSEMBLER MODULES

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

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

This causes some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers.

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

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

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

The first question is discussed in the section *Calling assembler routines from C*, page 125. The following two are covered in the section *Calling convention*, page 128.

For information about how data in memory is accessed, see *Memory access methods*, page 136.

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 139.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 125, and *Calling assembler routines from C++*, page 127, respectively.

## INLINE ASSEMBLER

Inline assembler can be used for inserting assembler instructions directly into a C or C++ function.

The `asm` extended keyword and its alias `__asm` both insert assembler instructions. However, when you compile C source code, the `asm` keyword is not available when the option `--strict` is used. The `__asm` keyword is always available.

**Note:** Not all assembler directives or operators can be inserted using these keywords.

The syntax is:

```
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:

```
asm("label:nop\n"
    "br label");
```

where `\n` (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions.

The following example demonstrates the use of the `asm` keyword. This example also shows the risks of using inline assembler.

```
extern volatile char UART1_SR;
#pragma required=UART1_SR

static char sFlag;

void loop_until_flag_is_set(void)
{
    while (!sFlag)
    {
        asm("LOADW 0x1000, R0");
        asm("STORW R0, sFlag:M");
    }
}
```

In this example, the assignment to the global variable `sFlag` 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 might 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.

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:

```
extern int foo(void);
or
extern int foo(int i, int j);
```

One way of fulfilling these requirements is to create skeleton code in C, compile it, and study the assembler list file.

### CREATING SKELETON CODE

The recommended way to create an assembler language routine with the correct interface is to start with an assembler language source file created by the C compiler. Note that you must create skeleton code for each function prototype.

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the variables required and perform simple accesses to them. In this example, the assembler routine takes an `int` and a `char`, and then returns an `int`:

```
extern int gInt;
extern char gChar;

int Func(int arg1, char arg2)
{
    int locInt = arg1;
    gInt = arg1;
    gChar = arg2;
    return locInt;
}

int main()
{
    int locInt = gInt;
    gInt = Func(locInt, gChar);
    return 0;
}
```

**Note:** In this example we use a low optimization level when compiling the code to show local and global variable access. If a higher level of optimization is used, the required

references to local variables could be removed during the optimization. The actual function declaration is not changed by the optimization level.

## COMPILING THE CODE



In the IDE, specify list options on file level. Select the file in the workspace window. Then choose **Project>Options**. In the **C/C++ Compiler** category, select **Override inherited settings**. On the **List** page, deselect **Output list file**, and instead select the **Output assembler file** option and its suboption **Include source**. Also, be sure to specify a low level of optimization.



Use these options to compile the skeleton code:

```
icccr16c 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 `s45`. Also remember to specify the code model and data model you are using, a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s45`.

**Note:** The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file.



In the IDE, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**.



On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.

## The output file

The output file contains the following important information:

- The calling convention
- The return values
- The global variables
- The function parameters
- How to create space on the stack (auto variables)
- Call frame information (CFI).

The CFI directives describe the call frame information needed by the Call Stack window in the debugger. For more information, see *Call frame information*, page 139.

## Calling assembler routines from C++

The C calling convention does not apply to C++ functions. Most importantly, a function name is not sufficient to identify a C++ function. The scope and the type of the function are also required to guarantee type-safe linkage, and to resolve overloading.

Another difference is that non-static member functions get an extra, hidden argument, the `this` pointer.

However, when using C linkage, the calling convention conforms to the C calling convention. An assembler routine can therefore be called from C++ when declared in this manner:

```
extern "C"
{
    int MyRoutine(int);
}
```

In C++, data structures that only use C features are known as PODs (“plain old data structures”), they use the same memory layout as in C. However, we do not recommend that you access non-PODs from assembler routines.

The following example shows how to achieve the equivalent to a non-static member function, which means that the implicit `this` pointer must be made explicit. It is also possible to “wrap” the call to the assembler routine in a member function. Use an inline member function to remove the overhead of the extra call—this assumes that function inlining is enabled:

```
class MyClass;

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

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

Note: Support for C++ names from assembler code is extremely limited. This means that:

- Assembler list files resulting from compiling C++ files cannot, in general, be passed through the assembler.

- It is not possible to refer to or define C++ functions that do not have C linkage in assembler.

---

## Calling convention

A calling convention is the way a function in a program calls another function. The compiler handles this automatically, but, if a function is written in assembler language, you must know where and how its parameters can be found, how to return to the program location from where it was called, and how to return the resulting value.

It is also important to know which registers an assembler-level routine must preserve. If the program preserves too many registers, the program might be ineffective. If it preserves too few registers, the result would be an incorrect program.

This section describes the calling convention used by the compiler. These items are examined:

- Function declarations
- C and C++ linkage
- Preserved versus scratch registers
- Function entrance
- Function exit
- Return address handling.

At the end of the section, some examples are shown to describe the calling convention in practice.

### FUNCTION DECLARATIONS

In C, a function must be declared in order for the compiler to know how to call it. A declaration could look as follows:

```
int MyFunction(int first, char * second);
```

This means that the function takes two parameters: an integer and a pointer to a character. The function returns a value, an integer.

In the general case, this is the only knowledge that the compiler has about a function. Therefore, it must be able to deduce the calling convention from this information.

### USING C LINKAGE IN C++ SOURCE CODE

In C++, a function can have either C or C++ linkage. To call assembler routines from C++, it is easiest if you make the C++ function have C linkage.



This is an example of a declaration of a function with C linkage:

```
extern "C"
{
    int F(int);
}
```

It is often practical to share header files between C and C++. This is an example of a declaration that declares a function with C linkage in both C and C++:

```
#ifdef __cplusplus
extern "C"
{
#endif

int F(int);

#ifdef __cplusplus
}
#endif
```

## PRESERVED VERSUS SCRATCH REGISTERS

The general CR16C 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 R6, and the return address registers, can be used as a scratch register by the function.

### 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 R7 through to R15, but not including the return address registers, are preserved registers.

## Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer register 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, will be destroyed.
- The global base pointer register `R13` (points to an area of data that is addressed with indexed addressing modes) must never be changed. In the eventuality of an interrupt, the register must have a specific value.
- The link register 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. The parameters are also passed on the stack in these cases:

- Structure types: `struct`, `union`, and classes
- The data type `double` (64-bit floating-point numbers)
- Unnamed parameters to variable length (variadic) functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

**Note:** Interrupt functions cannot take any parameters.

## 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 or a `double`, the memory location where the structure will be stored is passed in the register `R0` or in the register pair (`R1`, `R0`) as a hidden parameter, depending on the size of the default pointer.

## Register parameters

The registers available for passing parameters are `R2`, `R3`, `R4`, and `R5`.

Parameters	Passed in registers
8-bit scalar values	<code>R2</code> , <code>R3</code> , <code>R4</code> , or <code>R5</code>
16-bit scalar values	<code>R2</code> , <code>R3</code> , <code>R4</code> , or <code>R5</code>

Table 23: Registers used for passing parameters

Parameters	Passed in registers
32-bit scalar values	(R3, R2) or (R5, R4)

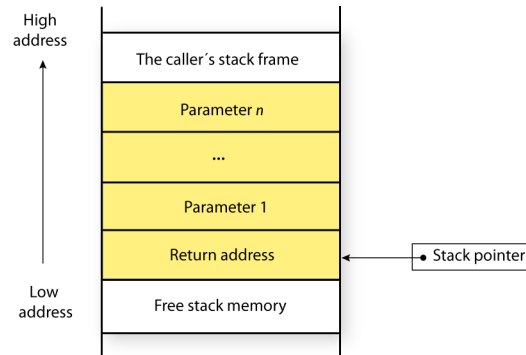
Table 23: Registers used for passing parameters (Continued)

The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order 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.

### Stack parameters and layout

Stack parameters are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer (toward 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 two, etc.

This figure illustrates how parameters are stored on the stack:



### 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 R1.

Return values	Returned in
8-bit scalar values	R0
16-bit scalar values	R0
32-bit scalar values	(R1, R0)
Structure and 64-bit double*	The location pointed to by R0 in the Small data model, or by (R1, R0) in the other data models

Table 24: Registers used for returning values

\* The called function must return the pointer in the register R0 or in (R1, R0), respectively.

## Stack layout at function exit

It is the responsibility of the caller to clean the stack after the called function returns.

## Return address handling

A function written in assembler language should, when finished, return to the caller. At a function call, the return address is stored in the return address register or registers. This is register RA in the normal code model, and the register pair (RA, R13) in the Short model.

Typically, a function returns by using the JUMP instruction, for example:

```
jump    (RA)
```

If a function is to call another function, the original return address must be stored somewhere. This is normally done on the stack, for example:

```

name    call
rseg    CODE:CODE
extern  func
public  call_func

call_func:
push    ra
bal     (ra), func

; Do something here.

popret  ra
end
```

The return address is restored directly from the stack with the `POPRET` instruction.

## RESTRICTIONS FOR SPECIAL FUNCTION TYPES

When trap functions are being called using the processor instruction `EXCP`, the return address will point to the instruction itself. To return to the instruction after the `EXCP` instruction, the return address will by default be adjusted within the trap function.

Also note that the processor uses different stacks in user and supervisor mode. This means that when calling functions that switch between the modes—such as user functions and possibly trap functions—the stack should not be used for passing parameters or return values. This is not checked by the compiler.

Interrupt and trap functions use the return instruction `RETX`.

## EXAMPLES

The following section shows a series of declaration examples and the corresponding calling conventions. The complexity of the examples increases toward the end.

### Example 1

Assume this function declaration:

```
int add1(int);
```

This function takes one parameter in the register `R2`, and the return value is passed back to its caller in the register `R0`.

This assembler routine is compatible with the declaration; it will return a value that is one number higher than the value of its parameter:

```
name      return
rseg      CODE:CODE
addw      $1, r2
movw      r2, r0
jump      (ra)
end
```

### Example 2

This example shows how structures are passed on the stack. Assume these declarations:

```
struct MyStruct
{
    int a;
    int b;
    int c;
    int d;
    int e;
};

int MyFunction(struct MyStruct x, int y);
```

The calling function must reserve 10 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register `R2`. The return value is passed back to its caller in the register `R0`.

### Example 3

The function below will return a structure of type `struct MyStruct`.

```
struct MyStruct
{
    int mA;
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in `R0` in the Small data model, and in `(R1, R0)` in the other models. The caller assumes that these registers remain untouched. The parameter `x` is passed in `R2`.

Assume that the function instead was declared to return a pointer to the structure:

```
struct MyStruct *MyFunction(int x);
```

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in `R2`, and the return value is returned in `R0` or `(R1, R0)`, depending on the data model.

## FUNCTION DIRECTIVES

**Note:** This type of directive is primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The IAR C/C++ Compiler for CR16C does not use static overlay, because it has no use for it.

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

For more information about the function directives, see the *IAR Assembler Reference Guide for CR16C*.

---

## Calling functions

Functions can be called in two fundamentally different ways—directly or via a function pointer. In this section we will discuss how both types of calls will be performed for each code model.

### ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

When calling a function via a function pointer, the `JAL` instruction is used. The following sections illustrate how the different code models perform function calls.

#### Normal code model

The normal function calling instruction is the branch-and-link instruction:

```
bal (RA), label
```

The location that the called function should return to (that is, the location immediately after this instruction) is stored in the link register, `RA`.

When a function call is made via a function pointer, this code will be generated:

```
jal (RA), (R1,R0) ; Make call
```

The address is stored in a register and is then used for calling the function.

#### Short code model

A direct call using this code model is simply:

```
bal (RA,R13), label
```

When a function call is made via a function pointer, this code will be generated:

```
jal (RA,R13), (R1,R0) ; Make call
```

The address is stored in a register and is then used for calling the function.

## Memory access methods

This section describes the different memory types presented in the chapter *Data storage*. In addition to just presenting the assembler code used for accessing data, this section will explain the reason behind the different memory types.

You should be familiar with the CR16C instruction set, in particular the different addressing modes used by the instructions that can access memory.

The IAR Assembler uses the convention that the size of a memory instruction is controlled by the suffixes `S`, `M`, and `L`. For example:

Suffix	Size	Instruction example
<code>S</code>	2 bytes	<code>loadb my_var:S(R1,R0), R2</code>
<code>M</code>	4 bytes	<code>loadb my_var:M(R1,R0), R2</code>
<code>L</code>	6 bytes	<code>loadb my_var:L(R1,R0), R2</code>

Table 25: Specifying the size of an assembler memory instruction

For each of the access methods described in the following sections, there are three examples:

- Accessing a global variable
- Accessing a global array using an unknown index
- Accessing a structure using a pointer.

These three examples can be illustrated by this C program:

```
char myVar;
char MyArr[10];

struct MyStruct
{
    long mA;
    char mB;
};

char Foo(int i, struct MyStruct *p)
{
    return myVar + MyArr[i] + p->mB;
}
```

### THE DATA16 MEMORY ACCESS METHOD

Data16 memory is located in the first 64 Kbytes of memory. This is the only memory type that can be accessed using 16-bit pointers and using a 16-bit index type. The advantage is that a single 16-bit register can be used, instead of a 32-bit register or register pair.



Also a 16-bit pointer only occupies 2 bytes when stored in memory.

### Examples

These examples access data16 memory in different ways:

```
loadb  x:M,r0 ; Access the global variable x.

loadb  y:L(r2),r1 ; Access an entry in
                ; the global array y.

loadb  4(r3),r0 ; Access through a pointer.
```

### THE DATA20 MEMORY ACCESS METHOD

The first Mbyte of the memory can be accessed using a wide range of addressing modes not available for the rest of the memory. The data20 memory type is designed to use these addressing modes.

Both the pointer and the index type have a size of 32 bits.

### Examples

These examples access data20 memory in different ways:

```
loadb  x:M,r0 ; Access the global variable x.

loadb  y(r2,r1),r1 ; Access an entry in
                ; the global array y.

loadb  4(r4,r3),r0 ; Access through a pointer
```

### THE DATA24 MEMORY ACCESS METHOD

The data24 memory access method can access the first 16 Mbytes of memory. The drawback of this access method is that only a few of the addressing modes can be used. This can result in larger and slower code compared with code accessing other types of data.

Both the pointer and the index type have a size of 32 bits.

## Examples

These examples access data24 memory in different ways:

```

loadb  x,r0      ; Access the global variable x.

movd   $y,(r6,r5) ; Access an entry in
                        ; the global array y.

addd   (r2,r1),(r6,r5)
loadb  0(r6,r5),r1

loadb  4(r4,r3),r0 ; Access through a pointer.

```

## THE DATA32 MEMORY ACCESS METHOD

The data32 memory access method can access the entire memory range. The drawback of this access method is that only indirect addressing via registers can be used. This can result in larger and slower code compared with code accessing other types of data.

Both the pointer and the index type have a size of 32 bits.

## Examples

These examples access data32 memory in different ways:

```

movd   $x,(r6,r5) ; Access the global
                        ; variable x.

loadb  0(r6,r5),r0

movd   $y,(r6,r5) ; Access an entry in
                        ; the global array y.

addd   (r2,r1),(r6,r5)
loadb  0(r6,r5),r1

loadb  4(r4,r3),r0 ; Access through a pointer.

```

## THE INDEX20 MEMORY ACCESS METHOD

Index20 memory is a 1-Mbyte area that can be placed anywhere in memory. The compiler assumes that one of the 32-bit registers, R13, permanently points to this area. The name index comes from the indexed addressing mode that is used to access this memory area.

The global label ?INDEX\_BASE is also assumed to be located at the beginning of this area.

The index20 memory access method places the difference between the location of the variable and the index base into the displacement field of the indexed addressing mode. Register R13 is used as the index register.

## Examples

These examples access index20 memory in different ways:

```
;; Access the global variable x.
loadb  [r13] (x-?INDEX_BASE), r0

;; Access an entry in the global array y.
loadb  [r13] (y-?INDEX_BASE) (r6, r5), r0

;; Access through a pointer.
loadb  4(r4, r3), r0
```

## THE INDEX4 AND INDEX5 MEMORY ACCESS METHODS

The index4 and index5 memory access methods can access 14 or 28 bytes, respectively, at the beginning of the same 1-Mbyte area as used by index20. The reason to use them is that more effective instructions can access the first bytes. For example:

```
;; Access the global variable x.
loadb  (x-?INDEX_BASE):s(R13), r0
```

---

## Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *IAR Assembler Reference Guide for CR16C*.

### CFI DIRECTIVES

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

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

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

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

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

Resource	Description
CFA_SP, CFA_ISP	The call frames of the regular stack and of the interrupt stack, respectively
R0–R13	Normal registers
RA	The return address register
SP	The stack pointer
PSR, CFG, INTBASE, USP, DSR, DCR	Special processor registers
?RET32	The return address shifted one step to the right (the same way that it is stored in RA); this is normally used only in the normal code model
?RET16H, ?RET16L	These resources describe the return address as two parts shifted one step to the right, the same way it is stored in the register pair (RA,ERA); they are useful in the small code model

Table 26: 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:

```
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 `-IA`.



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 cfi

RTMODEL "__code_model", "normal"
RTMODEL "__data_model", "indexed"
RTMODEL "__rt_version", "3"

RSEG CSTACK:DATA:SORT:NOROOT(0)
RSEG ISTACK:DATA:SORT:NOROOT(0)

PUBLIC cfiExample
FUNCTION cfiExample,021203H
ARGFRAME CSTACK, 0, STACK
LOCFRAME CSTACK, 6, STACK

CFI Names cfiNames0
CFI StackFrame CFA_SP SP DATA
CFI StackFrame CFA_ISP ISP DATA
CFI Resource R0:16, R1:16, R2:16, R3:16, R4:16, R5:16,
R6:16, R7:16
CFI Resource R8:16, R9:16, R10:16, R11:16, R12:32,
R13:32, RA:32, SP:32
CFI Resource PSR:32, CFG:32, INTBASE:32, ISP:32,
USP:32, DSR:32, DCR:32
CFI Resource DBS:32, CAR0:32, CAR1:32
CFI VirtualResource ?RET1:1, ?RET16L:16, ?RET16H:16,
?RET32:32, ?RET:33
CFI ResourceParts ?RET32 ?RET16H, ?RET16L
CFI ResourceParts ?RET ?RET32, ?RET1
CFI EndNames cfiNames0

```

```
CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 2
CFI DataAlign 2
CFI ReturnAddress ?RET CODE
CFI CFA_SP SP+0
CFI CFA_ISP ISP+0
CFI R0 Undefined
CFI R1 Undefined
CFI R2 Undefined
CFI R3 Undefined
CFI R4 Undefined
CFI R5 Undefined
CFI R6 Undefined
CFI R7 SameValue
CFI R8 SameValue
CFI R9 SameValue
CFI R10 SameValue
CFI R11 SameValue
CFI R12 SameValue
CFI R13 SameValue
CFI RA Undefined
CFI PSR Undefined
CFI CFG SameValue
CFI INTBASE SameValue
CFI USP SameValue
CFI DSR SameValue
CFI DCR SameValue
CFI DBS SameValue
CFI CAR0 SameValue
CFI CAR1 SameValue
CFI ?RET1 0
CFI ?RET16L Undefined
CFI ?RET16H Undefined
CFI ?RET32 RA
CFI ?RET Concat
CFI EndCommon cfiCommon0
```

```

        EXTERN F
        FUNCTION F,0202H

        RSEG CODE:CODE:NOROOT(0)
cfiExample:
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample
        FUNCALL cfiExample, F
        LOCFRAME CSTACK, 6, STACK
        PUSH    $1, R7, (RA)
        CFI ?RET32 Frame(CFA_SP, -4)
        CFI R7 Frame(CFA_SP, -6)
        CFI CFA_SP SP+6
        MOVW    R2, R7
        MOVW    R7, R2
        BAL     (RA), F
        ADDW    R7, R0
        POPRET  $1, R7, (RA)
        CFI EndBlock cfiBlock0

        END

```

**Note:** The header file `cfi.m45` contains the macros `XCFI_NAMES` and `XCFI_COMMON`, which declare a typical names block and a typical common block. These two macros declare several resources, both concrete and virtual.





# Using C

This chapter gives an overview of the compiler's support for the C language. The chapter also gives a brief overview of the IAR C language extensions.

---

## C language overview

The IAR C/C++ Compiler for CR16C supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

In addition, the compiler also supports the ISO 9899:1990 standard (including all technical corrigenda and addenda), also known as C94, C90, C89, and ANSI C. In this guide, this standard is referred to as *C89*. Use the `--c89` compiler option to enable this standard.

The C99 standard is derived from C89, but adds features like these:

- The `inline` keyword advises the compiler that the function defined immediately after the keyword should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a `for` loop
- The `bool` data type
- The `long long` data type
- The `complex` floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator `_Pragma()`
- Variadic macros, which are the preprocessor macro equivalents of `printf` style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option `--vla`
- Inline assembler using the `asm` or the `__asm` keyword, see *Inline assembler*, page 123.

**Note:**

- Even though it is a C99 feature, the IAR C/C++ Compiler for CR16C does not support UCNs (universal character names).
- CLIB does not support any C99 functionality. For example, complex numbers and variable length arrays are not supported.

---

## Extensions overview

The compiler offers the features of Standard C and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

This is an overview of the available extensions:

- IAR C language extensions

For information about available language extensions, see *IAR C language extensions*, page 147. For more information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

The `#pragma` directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages.

Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For information about available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

The preprocessor of the compiler adheres to Standard C. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. For more information about using intrinsic functions, see *Mixing C and*

*assembler*, page 121. For information about available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides the C and C++ library definitions that apply to embedded systems. For more information, see *IAR DLIB Library*, page 285.

**Note:** Any use of these extensions, except for the pragma directives, makes your source code inconsistent with Standard C.

## ENABLING LANGUAGE EXTENSIONS

You can choose different levels of language conformance by means of project options:

Command line	IDE*	Description
<code>--strict</code>	<b>Strict</b>	All IAR C language extensions are disabled; errors are issued for anything that is not part of Standard C.
None	<b>Standard</b>	All <i>extensions to Standard C</i> are enabled, but no <i>extensions for embedded systems programming</i> . For information about extensions, see <i>IAR C language extensions</i> , page 147.
<code>-e</code>	<b>Standard with IAR extensions</b>	All <i>IAR C language extensions</i> are enabled.

Table 27: Language extensions

\* In the IDE, choose **Project>Options>C/C++ Compiler>Language>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

## IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific microprocessor you are using, typically to meet memory restrictions
- *Relaxations to Standard C*—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions, see *Relaxations to Standard C*, page 150.

## EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

The following language extensions are available both in the C and the C++ programming languages and they are well suited for embedded systems programming:

- Memory attributes, 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 segment

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 segment. For more information about using these features, see *Controlling data and function placement in memory*, page 173, and *location*, page 258.

- Alignment control

Each data type has its own alignment; for more information, see *Alignment*, page 227. If you want to change the alignment, the `#pragma pack`, and the `#pragma data_alignment` directives are available. If you want to check the alignment of an object, use the `__ALIGNOF__()` operator.

The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

- `__ALIGNOF__(type)`
- `__ALIGNOF__(expression)`

In the second form, the expression is not evaluated.

- Anonymous structs and unions

C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 171.

- Bitfields and non-standard types

In Standard C, a bitfield must be of the type `int` or `unsigned int`. Using IAR C language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. For more information, see *Bitfields*, page 229.

- `static_assert()`

The construction `static_assert(const-expression, "message");` can be used in C/C++. The construction will be evaluated at compile time and if `const-expression` is false, a message will be issued including the `message` string.

- Parameters in variadic macros

Variadic macros are the preprocessor macro equivalents of `printf` style functions. The preprocessor accepts variadic macros with no arguments, which means if no parameter matches the `...` parameter, the comma is then deleted in the `" , ##__VA_ARGS__"` macro definition. According to Standard C, the `...` parameter must be matched with at least one argument.

## Dedicated segment operators

The compiler supports getting the start address, end address, and size for a segment with these built-in segment operators:

<code>__segment_begin</code>	Returns the address of the first byte of the named segment.
<code>__segment_end</code>	Returns the address of the first byte <i>after</i> the named segment.
<code>__segment_size</code>	Returns the size of the named segment in bytes.

**Note:** The aliases `__segment_begin/__sfb`, `__segment_end/__sfe`, and `__segment_size/__sfs` can also be used.

The operators can be used on named segments defined in the linker configuration file.

These operators behave syntactically as if declared like:

```
void * __segment_begin(char const * segment)
void * __segment_end(char const * segment)
size_t * __segment_size(char const * segment)
```

When you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment in the linker configuration file, the segment operators can be used for getting the start and end address of the memory range where the segments were placed.

The named *segment* must be a string literal and it must have been declared earlier with the `#pragma segment` directive. If the segment was declared with a memory attribute *memattr*, the type of the `__segment_begin` operator is a pointer to *memattr* `void`. Otherwise, the type is a default pointer to `void`. Note that you must enable language extensions to use these operators.

### Example

In this example, the type of the `__segment_begin` operator is `void __data32 *`.

```
#pragma segment="MYSEGMENT" __data32
...
segment_start_address = __segment_begin("MYSEGMENT");
```

See also *segment*, page 264, and *location*, page 258.

## RELAXATIONS TO STANDARD C

This section lists and briefly describes the relaxation of some Standard C issues and also some useful but minor syntax extensions:

- Arrays of incomplete types
 

An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).
- Forward declaration of `enum` types
 

The extensions allow you to first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.
- Accepting missing semicolon at the end of a `struct` or `union` specifier
 

A warning—instead of an error—is issued if the semicolon at the end of a `struct` or `union` specifier is missing.
- Null and `void`

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In Standard C, some operators allow this kind of behavior, while others do not allow it.
- Casting pointers to integers in static initializers
 

In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 233.
- Taking the address of a register variable
 

In Standard C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.
- `long float` means `double`

The type `long float` is accepted as a synonym for `double`.
- Repeated `typedef` declarations
 

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.
- Mixing pointer types
 

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.

- Non-top level `const`  
Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.
- Non-lvalue arrays  
A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.
- Comments at the end of preprocessor directives  
This extension, which makes it legal to place text after preprocessor directives, is enabled unless the strict Standard C mode is used. The purpose of this language extension is to support compilation of legacy code; we do not recommend that you write new code in this fashion.
- An extra comma at the end of `enum` lists  
Placing an extra comma is allowed at the end of an `enum` list. In strict Standard C mode, a warning is issued.
- A label preceding a `}`  
In Standard C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler allows this, but issues a warning.  
Note that this also applies to the labels of `switch` statements.
- Empty declarations  
An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).
- Single-value initialization  
Standard C requires that all initializer expressions of static arrays, structs, and unions are enclosed in braces.  
Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:  

```
struct str
{
    int a;
} x = 10;
```

- Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

- Expanding function names into strings with the function as context

Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string that contains the name of the current function. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char)"
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 208.

- Static functions in function and block scopes

Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- Numbers scanned according to the syntax for numbers

Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one valid token. (If the `--strict` option is used, the `pp-number` syntax is used instead.)



# Using C++

IAR Systems supports the C++ language. You can choose between the industry-standard Embedded C++ and Extended Embedded C++. This chapter describes what you need to consider when using the C++ language.

---

## Overview

Embedded C++ is a proper 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. EC++ offers the same object-oriented benefits as C++, but without some features that can increase code size and execution time in ways that are hard to predict.

### EMBEDDED C++

These C++ features are supported:

- Classes, which are user-defined types that incorporate both data structure and behavior; the essential feature of inheritance allows data structure and behavior to be shared among classes
- Polymorphism, which means that an operation can behave differently on different classes, is provided by virtual functions
- Overloading of operators and function names, which allows several operators or functions with the same name, provided that their argument lists are sufficiently different
- Type-safe memory management using the operators `new` and `delete`
- Inline functions, which are indicated as particularly suitable for inline expansion.

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are features added very late before Standard C++ was defined. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

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

- Templates
- Multiple and virtual inheritance
- Exception handling
- Runtime type information

- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

**Note:** The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

### EXTENDED EMBEDDED C++

IAR Systems' Extended EC++ is a slightly larger subset of C++ which adds these features to the standard EC++:

- Full template support
- 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 support for C++



In the compiler, the default language is C.

To compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 208.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 209.

For EC++, and EEC++, you must also use the IAR DLIB runtime library.



To enable EC++ or EEC++ in the IDE, choose **Project>Options>C/C++ Compiler>Language** and select the appropriate standard.

---

## EC++ feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for CR16C, you must be aware of some benefits and some possible quirks when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

### USING IAR ATTRIBUTES WITH CLASSES

Static data members of C++ classes are treated the same way global variables are, and can have any applicable IAR type, memory, and object attribute.

Member functions are in general treated the same way free functions are, and can have any applicable IAR type and object attributes. Virtual member functions can only have attributes that are compatible with default function pointers, and constructors and destructors cannot have any such attributes.

The location operator `@` and the `#pragma location` directive can be used on static data members and with all member functions.

### Example

```
class MyClass
{
public:
    // Locate a static variable in data16 memory at address 60
    static __data16 __no_init int mI @ 60;

    // Declare static member function using __trap.
    #pragma vector=4
    static __trap void F();

    // Declare member function using __trap
    #pragma vector=8
    __trap void G();

    // Locate a virtual function into SPECIAL
    virtual void M() const volatile @ "SPECIAL";
};
```

### The `this` pointer

The `this` pointer used for referring to a class object or calling a member function of a class object will by default have the data memory attribute for the default data pointer type. This means that such a class object can only be defined to reside in memory from which pointers can be implicitly converted to a default data pointer. This restriction might also apply to objects residing on a stack, for example temporary objects and auto objects.

### Class memory

To compensate for this limitation, a class can be associated with a *class memory type*. The class memory type changes:

- the `this` pointer type in all member functions, constructors, and destructors into a pointer to class memory
- the default memory for static storage duration variables—that is, not auto variables—of the class type, into the specified class memory
- the pointer type used for pointing to objects of the class type, into a pointer to class memory.

**Example**

```

class __data24 C
{
public:
    void MyF();           // Has a this pointer of type C __data24 *
    void MyF() const;    // Has a this pointer of type
                        // C __data24 const *
    C();                 // Has a this pointer pointing into Data24
                        // memory
    C(C const &);        // Takes a parameter of type C __data24
                        // const & (also true of generated copy
                        // constructor)

    int mI;
};

C Ca;                   // Resides in Data24 memory instead of the
                        // default memory
C __data20 Cb;          // Resides in Data20 memory, the 'this'
                        // pointer still points into Data24 memory

void MyH()
{
    C cd;                // Resides on the stack
}

C *Cp1;                 // Creates a pointer to Data24 memory
C __data20 *Cp2;        // Creates a pointer to Data20 memory

```

Whenever a class type associated with a class memory type, like `C`, must be declared, the class memory type must be mentioned as well:

```
class __data20 C;
```

Also note that class types associated with different class memories are not compatible types.

A built-in operator returns the class memory type associated with a class, `__memory_of(class)`. For instance, `__memory_of(C)` returns `__data20`.

When inheriting, the rule is that it must be possible to convert implicitly a pointer to a subclass into a pointer to its base class. This means that a subclass can have a *more* restrictive class memory than its base class, but not a *less* restrictive class memory.

```

class __data24 D : public C
{ // OK, same class memory
public:
    void MyG();
    int mJ;
};

class __data20 E : public C
{ // OK, data20 memory is inside data24
public:
    void MyG() // Has a this pointer pointing into data20 memory
    {
        MyF(); // Gets a this pointer into data24 memory
    }
    int mJ;
};

class F : public C
{ // OK, will be associated with same class memory as C
public:
    void MyG();
    int mJ;
};

```

Note that the following is not allowed because data24 is not inside data20 memory:

```

class __data24 G:public C
{
};

```

A new expression on the class will allocate memory in the heap associated with the class memory. A delete expression will naturally deallocate the memory back to the same heap. To override the default new and delete operator for a class, declare

```

void *operator new(size_t);
void operator delete(void *);

```

as member functions, just like in ordinary C++.

For more information about memory types, see *Memory types*, page 46.

## 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
}
```

## NEW AND DELETE OPERATORS

There are operators for `new` and `delete` for each memory that can have a heap, that is, `data16` and `data32` memory.

```
#include <stddef.h>

// Assumes a heap in both __data16 and __data32 memory
void __data16 * operator new __data16(__data16_size_t);
void __data32 * operator new __data32(__data32_size_t);
void operator delete(void __data16 *);
void operator delete(void __data32 *);

// And correspondingly for array new and delete operators
void __data16 * operator new[] __data16(__data16_size_t);
void __data32 * operator new[] __data32(__data32_size_t);
void operator delete[](void __data16 *);
void operator delete[](void __data32 *);
```

Use this syntax if you want to override both global and class-specific operator `new` and operator `delete` for any data memory.

Note that there is a special syntax to name the operator `new` functions for each memory, while the naming for the operator `delete` functions relies on normal overloading.

## New and delete expressions

A new expression calls the `operator new` function for the memory of the type given. If a class, struct, or union type with a class memory is used, the class memory will determine the `operator new` function called. For example,

```
void MyF()
{
    // Calls operator new __data16(__data16_size_t)
    int __data16 *p = new __data16 int;

    // Calls operator new __data16(__data16_size_t)
    int __data16 *q = new int __data16;

    // Calls operator new[] __data16(__data16_size_t)
    int __data16 *r = new __data16 int[10];

    // Calls operator new __data32(__data32_size_t)
    class __data32 S
    {
    };
    S *s = new S;

    // Calls operator delete(void __data16 *)
    delete p;
    // Calls operator delete(void __data32 *)
    delete s;

    int __data32 *t = new __data16 int;
    delete t; // Error: Causes a corrupt heap
}
```

Note that the pointer used in a `delete` expression must have the correct type, that is, the same type as that returned by the `new` expression. If you use a pointer to the wrong memory, the result might be a corrupt heap.

## USING STATIC CLASS OBJECTS IN INTERRUPTS

If interrupt functions use static class objects that need to be constructed (using constructors) or destroyed (using destructors), your application will not work properly if the interrupt occurs before the objects are constructed, or, during or after the objects are destroyed.

To avoid this, make sure that these interrupts are not enabled until the static objects have been constructed, and are disabled when returning from `main` or calling `exit`. For information about system startup, see *System startup and termination*, page 92.



Function local static class objects are constructed the first time execution passes through their declaration, and are destroyed when returning from `main` or when calling `exit`.

## USING NEW HANDLERS

To handle memory exhaustion, you can use the `set_new_handler` function.

### New handlers in Embedded C++

If you do not call `set_new_handler`, or call it with a `NULL` new handler, and `operator new` fails to allocate enough memory, it will call `abort`. The `nothrow` variant of the new operator will instead return `NULL`.

If you call `set_new_handler` with a non-`NULL` new handler, the provided new handler will be called by `operator new` if `operator new` fails to allocate memory. The new handler must then make more memory available and return, or abort execution in some manner. The `nothrow` variant of `operator new` will never return `NULL` in the presence of a new handler.

## TEMPLATES

Extended EC++ supports templates according to the C++ standard, but not the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

## DEBUG SUPPORT IN C-SPY

C-SPY has 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.

For more information about this, see the *C-SPY® Debugging Guide*.

---

## EEC++ feature description

This section describes features that distinguish Extended EC++ from EC++.

### TEMPLATES

The compiler supports templates with the syntax and semantics as defined by Standard C++. However, note that the STL (standard template library) delivered with the product is tailored for Extended EC++, see *Extended Embedded C++*, page 154.

## Templates and data memory attributes

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling were changed—class template partial specialization matching and function template parameter deduction.

In Extended Embedded C++, the class template partial specialization matching algorithm works like this:

*When a pointer or reference type is matched against a pointer or reference to a template parameter type, the template parameter type will be the type pointed to, stripped of any data memory attributes, if the resulting pointer or reference type is the same.*

### Example

```
// We assume that __data24 is the memory type of the default
// pointer.
template<typename> class Z {};
template<typename T> class Z<T *> {};

Z<int __data16 *> Zn;    // T = int __data16
Z<int __data24 *> Zf;    // T = int
Z<int          *> Zd;    // T = int
Z<int __data32 *> Zh;    // T = int __data32
```

In Extended Embedded C++, the function template parameter deduction algorithm works like this:

*When function template matching is performed and an argument is used for the deduction; if that argument is a pointer to a memory that can be implicitly converted to a default pointer, do the parameter deduction as if it was a default pointer.*

*When an argument is matched against a reference, do the deduction as if the argument and the parameter were both pointers.*

**Example**

```
// We assume that __data24 is the memory type of the default
// pointer.
template<typename T> void fun(T *);

void MyF()
{
    fun((int __data20 *) 0); // T = int. The result is different
                            // than the analogous situation with
                            // class template specializations.
    fun((int          *) 0); // T = int
    fun((int __data24 *) 0); // T = int
    fun((int __data32 *) 0); // T = int __data32
}
```

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to “small” memory types. For “large” and “other” memory types (memory that cannot be pointed to by a default pointer) it is possible. To make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

**Example**

```
// We assume that __data24 is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
    fun((int __data16 *) 0); // T = int __data16
}
```

**Non-type template parameters**

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

**Example**

```
extern int __data20 X;

template<__data20 int &y>
void my_set_template()
{
    y = 17;
}

void my_set_x()
{
    my_set_template<X>();
}
```

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

**Example**

```
#include <vector>

vector<int> D; // D placed in default memory
              // using the default heap,
              // uses default pointers

vector<int __data20> __data20 X; // X placed in data20 memory,
                                  // heap allocation from
                                  // data20, uses pointers to
                                  // data20 memory

vector<int __data32> __data20 Y; // Y placed in data20 memory,
                                  // heap allocation from
                                  // data32, uses pointers to
                                  // data32 memory
```

Note that this is illegal:

```
vector<int __data16> __data32 Z;
```

Note also that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T> mem` where `mem` is the memory type of `T`. Supplying a key with a memory type is not useful.

**Example**

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated assign member method must be used.

```
#include <vector>

vector<int __data20> X;
vector<int __data32> Y;

void MyF()
{
    // The templated assign member method will work
    X.assign(Y.begin(), Y.end());
    Y.assign(X.begin(), X.end());
}
```

**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.

## C++ language extensions

When you use the compiler in any C++ mode and enable IAR language extensions, the following C++ language extensions are available in the compiler:

- In a friend declaration of a class, the `class` keyword can be omitted, for example:

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

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

```
class A
{
    const int mSize = 10; //Possible when using IAR language
                          //extensions
    int mArr[mSize];
};
```

According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:

```
struct A
{
    int A::F(); // Possible when using IAR language extensions
    int G();   // According to the standard
};
```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

```
extern "C" void F(); // Function with C linkage
void (*PF)()        // PF points to a function with C++ linkage
                   = &F; // Implicit conversion of function pointer.
```

According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```
bool X;

char *P1 = X ? "abc" : "def";           //Possible when using IAR
                                        //language extensions
char const *P2 = X ? "abc" : "def"; //According to the standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.
- An anonymous union can be introduced into a containing class by a `typedef` name. It is not necessary to first declare the union. For example:

```
typedef union
{
    int i,j;
} U; // U identifies a reusable anonymous union.

class A
{
public:
    U; // OK -- references to A::i and A::j are allowed.
};
```

In addition, this extension also permits *anonymous classes* and *anonymous structs*, as long as they have no C++ features (for example, no static data members or member functions, and no non-public members) and have no nested types other than other anonymous classes, structs, or unions. For example:

```
struct A
{
    struct
    {
        int i,j;
    }; // OK -- references to A::i and A::j are allowed.
};
```

- The friend class syntax allows nonclass types as well as class types expressed through a `typedef` without an elaborated type name. For example:

```
typedef struct S ST;

class C
{
public:
    friend S; // Okay (requires S to be in scope)
    friend ST; // Okay (same as "friend S;")
    // friend S const; // Error, cv-qualifiers cannot
                       // appear directly
};
```

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



# 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 small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- Try to avoid the 64-bit data type `double`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.

- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array. For data16 this is `int`, and for all other memory types it is `long`.
- Using floating-point types on a microprocessor without a math co-processor is very inefficient, both in terms of code size and execution speed.

For information about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

## FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

In the compiler, the floating-point type `float` always uses the 32-bit format, and the type `double` always uses the 64-bit format.

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

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

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

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

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

For more information about floating-point types, see *Floating-point types*, page 231.

## ALIGNMENT OF ELEMENTS IN A STRUCTURE

The CR16C microprocessor requires that data in memory must be aligned. Each element in a structure must be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment correct.

There are situations when this can be a problem:

- There are external demands; for example, network communication protocols are usually specified in terms of data types with no padding in between
- You need to save data memory.

For information about alignment requirements, see *Alignment*, page 227.

There are two ways to solve the problem:

- Use the `#pragma pack` directive for a tighter layout of the structure. The drawback is that each access to an unaligned element in the structure will use more code.
- Write your own customized functions for *packing* and *unpacking* structures. This is a more portable way, which will not produce any more code apart from your functions. The drawback is the need for two views on the structure data—packed and unpacked.

For more information about the `#pragma pack` directive, see *pack*, page 261.

## 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 CR16C 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 208, for additional information.

### Example

In this example, the members in the anonymous union can be accessed, in function `F`, without explicitly specifying the union name:

```
struct S
{
    char mTag;
    union
    {
        long mL;
        float mF;
    };
} St;

void F(void)
{
    St.mL = 5;
}
```

The member names must be unique in the surrounding scope. Having an anonymous struct or union at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in this example:

```
__no_init volatile
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char way: 1;
        unsigned char out: 1;
    };
} @ 0xFF4900;

/* The variables are used here. */
void Test(void)
{
    IOPORT = 0;
    way = 1;
    out = 1;
}
```

This declares an I/O register byte `IOPORT` at address `0xFF4900`. 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:

- Code and data models

Use the different compiler options for code and data models, respectively, to take advantage of the different addressing modes available for the microprocessor and thereby also place functions and data objects in different parts of memory. For more information about data and code models, see *Data models*, page 44, and *Code models and register modes*, page 55, respectively.

- Memory attributes

Use memory attributes to override the default addressing mode and placement of data objects. For more information about memory attributes for data, see *Using data memory attributes*, page 47.

- 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 either `__no_init` or `const`.

This is useful for individual data objects that must be located at a fixed address to conform to external requirements, for example to populate interrupt vectors or other hardware tables. Note that it is not possible to use this notation for absolute placement of individual functions.

- The `@` operator and the `#pragma location` directive for segment placement

Use the `@` operator or the `#pragma location` directive to place groups of functions or global and static variables in named segments, without having explicit control of each object. The variables must be declared either `__no_init` or `const`. The segments can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the `segment 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 segments when absolute control over the placement of individual variables is not needed, or not useful.

- The `--segment` option

Use the `--segment` option to place functions and/or data objects in named segments, which is useful, for example, if you want to direct them to different fast or slow memories. In contrast to using the `@` operator or the `#pragma location` directive, there are no restrictions on what types of variables that can be placed in named segments when using the `--segment` option. For more information, see *--segment*, page 222.

At compile time, data and functions are placed in different segments, see *Data segments*, page 69, and *Code segments*, page 75, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker configuration file, see *Placing segments in memory*, page 66.

## 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)
- `const` (with 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:** All declarations of variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

### Examples

In this example, a `__no_init` declared variable is placed at an absolute address. This is useful for interfacing between multiple processes, applications, etc:

```
__no_init volatile char alpha @ 0xFF2000; /* OK */
```

The next example contains two `const` declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM.

This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0xFF2002
__no_init const int beta;           /* OK */

const int gamma @ 0xFF2004 = 3;     /* OK */
```

In the first case, the value is not initialized by the compiler; the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

These examples show incorrect usage:

```
int delta @ 0xFF2006;               /* Error, neither */
                                   /* "__no_init" nor "const".*/

__no_init int epsilon @ 0xFF2007;  /* Error, misaligned. */
```

### C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```
volatile const __no_init int x @ 0x100;          /* Bad in C++ */
```

the linker will report that more than one variable is located at address 0x100.

To avoid this problem and make the process the same in C and C++, you should declare these variables `extern`, for example:

```
/* The extern keyword makes x public. */
extern volatile const __no_init int x @ 0x100;
```

**Note:** C++ static member variables can be placed at an absolute address just like any other static variable.

## DATA AND FUNCTION PLACEMENT IN SEGMENTS

The following method can be used for placing data or functions in named segments other than default:

The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. The named segment can either be a predefined segment, or a user-defined segment. The variables must be declared either `__no_init` or `const`. If declared `const`, they can have initializers.

The `--segment` option can be used for placing variables and functions, which are parts of the whole compilation unit, in named segments.

C++ static member variables can be placed in named segments just like any other static variable.

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker configuration file using the `-Z` or the `-P` segment control directives.

**Note:** Take care when explicitly placing a variable or function in a predefined segment 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 segments can be controlled from the linker configuration file.

For more information about segments, see the chapter *Segment reference*.

### Examples of placing variables in named segments

In the following examples, a data object is placed in a user-defined segment. If no memory attribute is specified, the variable will, like any other variable, be treated as if it is located in the default memory. Note that you must place the user-defined segment appropriately in the linker configuration file.

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

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

const int gamma @ "MY_CONSTANTS" = 17; /* OK */
```

As usual, you can use memory attributes to select a memory for the variable. Note that you must as always place the segment appropriately in the linker configuration file.

```
__data32 __no_init int alpha @ "MY_DATA32_NOINIT"; /* Placed in
                                                    data32*/
```

This example shows incorrect usage:

```
int delta @ "MY_ZEROS"; /* Error, neither */
                        /* "__no_init" nor "const" */
```



### Examples of placing functions in named segments

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

#pragma location="MY_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. See *optimize*, page 260, 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, cross call, 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 213.

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 206.

## OPTIMIZATION LEVELS

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

Optimization level	Description
None (Best debug support)	Variables live through their entire scope Dead code elimination Redundant label elimination Redundant branch elimination
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope
Medium	Same as above, and: Live-dead analysis and optimization Code hoisting Register content analysis and optimization Common subexpression elimination Code motion
High (Balanced)	Same as above, and: Peephole optimization Cross jumping Cross call Loop unrolling Function inlining Code motion Type-based alias analysis

Table 28: Compiler optimization levels

**Note:** Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 179.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers

used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

## SPEED VERSUS SIZE

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

If you use the optimization level High speed, the `--no_size_constraints` compiler option relaxes the normal restrictions for code size expansion and enables more aggressive optimizations.

## FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--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
- Cross call

### Common subexpression elimination

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

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

For more information about the command line option, see `--no_cse`, page 214.

## Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most efficient for smaller loops, where the loop overhead can be a substantial part of the total loop body.

Loop unrolling, which can be performed at optimization level High, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

The compiler heuristically decides which loops to unroll. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

To disable loop unrolling, use the command line option `--no_unroll`, see `--no_unroll`, page 216.

## Function inlining

Function inlining means that a function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization normally reduces execution time, but might increase the code size.

For more information, see *Inlining functions*, page 63.

## Code motion

Evaluation of loop-invariant expressions and common subexpressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization level High, normally reduces code size and execution time. The resulting code might however be difficult to debug.

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

For more information about the command line option, see `--no_code_motion`, page 213.

## Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object are performed using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers can reference the same memory location or not.

Type-based alias analysis is performed at optimization level High. For application code conforming to standard C or C++ application code, this optimization can reduce code size and execution time. However, non-standard C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

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

For more information about the command line option, see `--no_tbaa`, page 216.

### Example

```
short F(short *p1, long *p2)
{
    *p2 = 0;
    *p1 = 1;
    return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. If you use explicit casts, you can also force pointers of different pointer types to point to the same memory location.

### Cross call

Common code sequences are extracted to local subroutines. This optimization, which is performed at optimization level High, can reduce code size, sometimes dramatically, on behalf of execution time and stack size. The resulting code might however be difficult to debug.

---

## Facilitating good code generation

This section contains hints on how to help the compiler generate good code, for example:

- Using efficient addressing modes
- Helping the compiler optimize
- Generating more useful error message.

## WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

The following is a list of programming techniques that will, when followed, enable the compiler to better optimize the application.

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables (non-static). Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 180. To maximize the effect of the inlining transformation, it is good practice to place the definitions of small functions called from more than one module in the header file rather than in the implementation file. Alternatively, you can use multi-file compilation. For more information, see *Multi-file compilation units*, page 177.
- Avoid using inline assembler. Instead, try writing the code in C/C++, use intrinsic functions, or write a separate module in assembler language. For more information, see *Mixing C and assembler*, page 121.

## SAVING STACK SPACE AND RAM MEMORY

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

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

## FUNCTION PROTOTYPES

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

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

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

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

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

### Prototyped style

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

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

int Test(char ch, int i) /* Definition */
{
    return i + ch;
}
```

### Kernighan & Ritchie style

In K&R style—pre-Standard C—it is not possible to declare a function prototyped. Instead, an empty parameter list is used in the function declaration. Also, the definition looks different.

For example:

```
int Test(); /* Declaration */

int Test(ch, i) /* Definition */
char ch;
int i;
{
    return i + ch;
}
```

## INTEGER TYPES AND BIT NEGATION

In some situations, the rules for integer types and their conversion lead to possibly confusing behavior. Things to look out for are assignments or conditionals (test expressions) involving types with different size, and logical operations, especially bit negation. Here, *types* also includes types of constants.

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 16-bit integer, and two's complement is assumed:

```
void F1(unsigned char c1)
{
    if (c1 == ~0x80)
        ;
}
```

Here, the test is always false. On the right hand side, `0x80` is `0x0080`, and `~0x0080` becomes `0xFF7F`. 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 8 bits set.

## PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code.

For sequences of accesses to variables that you do not want to be interrupted, use the `__monitor` keyword. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. Accessing a small-sized `volatile` variable can be an atomic operation, but you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation. For more information, see *\_\_monitor*, page 247.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 236.



### Protecting the eeprom write mechanism

A typical example of when it can be necessary to use the `__monitor` keyword is when protecting the eeprom write mechanism, which can be used from two threads (for example, main code and interrupts). Servicing an interrupt during an EEPROM write sequence can in many cases corrupt the written data.



## ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several CR16C 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 `iolmx5100.h`:

```
__no_init volatile union
{
    unsigned short mwctl2;
    struct
    {
        unsigned short edr: 1;
        unsigned short edw: 1;
        unsigned short lee: 2;
        unsigned short lemd: 2;
        unsigned short lepl: 2;
    } mwctl2bit;
} @ 0xFF4900;

/* By including the appropriate include file in your code,
 * it is possible to access either the whole register or any
 * individual bit (or bitfields) from C code as follows.
 */

void Test()
{
    /* Whole register access */
    mwctl2 = 0x1234;

    /* Bitfield accesses */
    mwctl2bit.edw = 1;
    mwctl2bit.lepl = 3;
}
```

You can also use the header files as templates when you create new header files for other CR16C devices. For information about the @ operator, see *Located data*, page 75

## 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 segment, according to the specified memory keyword. See the chapter *Placing code and data* for more information.

For `__no_init`, the `const` keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

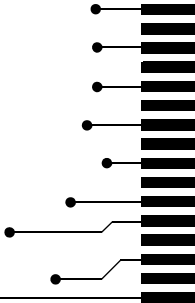
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

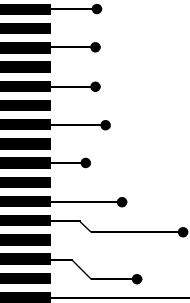
For more information, see *\_\_no\_init*, page 247. Note that to use this keyword, language extensions must be enabled; see *-e*, page 208. For more information, see also *object\_attribute*, page 259.

# Part 2. Reference information

This part of the *IAR C/C++ Compiler Reference Guide for CR16C* contains these chapters:

- External interface details
- External interface details
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- Segment reference
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89.





# External interface details

This chapter provides reference information about how the compiler interacts with its 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 output.

---

## Invocation syntax

You can use the compiler either from the IDE or from the command line. See the *IDE Project Management and Building Guide* for information about using the compiler from the IDE.

### COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
icccr16c [options] [sourcefile] [options]
```

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

```
icccr16c prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is not significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order as they are specified on the command line.

If you run the compiler from the command line without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

### PASSING OPTIONS

There are three different ways of passing options to the compiler:

- Directly from the command line  
Specify the options on the command line after the `icccr16c` command, either before or after the source filename; see *Invocation syntax*, page 189.

- Via environment variables  
The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 190.
- Via a text file, using the `-f` option; see *-f*, page 210.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *External interface details* chapter.

## ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: C_INCLUDE=c:\program files\iar systems\embedded workbench 6.n\cr16c\inc;c:\headers
QCCCR16C	Specifies command line options; for example: QCCCR16C=-lA asm.lst

Table 29: Compiler environment variables

## Include file search procedure

This is a detailed description of the compiler's `#include` file search procedure:

- If the name of the `#include` file is an absolute path specified in angle brackets or double quotes, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:  
`#include <stdio.h>`  
it searches these directories for the file to include:
  - 1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 211.
  - 2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 190.
  - 3 The automatically set up library system include directories. See *--clib*, page 201, *--dlib*, page 207, and *--dlib\_config*, page 207.
- If the compiler encounters the name of an `#include` file in double quotes, for example:  
`#include "vars.h"`  
it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.

If there are nested `#include` files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
#include "src.h"
...
src.h in directory dir\include
#include "config.h"
...
```

When `dir\exe` is the current directory, use this command for compilation:

```
icccr16c ..\src\src.c -I..\include -I..\debugconfig
```

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

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

Use angle brackets for standard header files, like `stdio.h`, and double quotes for files that are part of your application.

For information about the syntax for including header files, see *Overview of the preprocessor*, page 277.

---

## Compiler output

The compiler can produce the following output:

- A linkable object file
 

The object files produced by the compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `.obj`.
- Optional list files
 

Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 211. By default, these files will have the filename extension `.lst`.
- Optional preprocessor output files
 

A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `.i`.

- Diagnostic messages

Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 192.

- 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 192.

- 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 returns status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

Code	Description
0	Compilation successful, but there might have been warnings.
1	Warnings were produced and the option <code>--warnings_affect_exit_code</code> was used.
2	Errors occurred.
3	Fatal errors occurred, making the compiler abort.
4	Internal errors occurred, making the compiler abort.

Table 30: Error return codes

---

## Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.



## MESSAGE FORMAT

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

```
filename, line number level [tag]: message
```

with these elements:

<i>filename</i>	The name of the source file in which the issue was encountered
<i>line number</i>	The line number at which the compiler detected the issue
<i>level</i>	The level of seriousness of the issue
<i>tag</i>	A unique tag that identifies the diagnostic message
<i>message</i>	An explanation, possibly several lines long

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

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

## SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

### Remark

A diagnostic message that is produced when the compiler finds a source code 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 221.

### Warning

A diagnostic message that is produced when the compiler finds a potential programming error or omission which is of concern, but which does not prevent completion of the compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see page 217.

### Error

A diagnostic message that is produced when the compiler finds a construct which clearly violates the C or C++ language rules, such that code cannot be produced. 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 of the source code 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 197, for information about the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for information about the pragma directives that are available for setting severity levels.

### INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler. 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, which can be seen in the header of the list files generated by the compiler
- Your license number
- The exact internal error message text
- The source file of the application that generated the internal error
- A list of the options that were used when the internal error occurred.

# Compiler options

This chapter describes the syntax of compiler options and the general syntax rules for specifying option parameters, and gives detailed reference information about each option.

---

## Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IDE.



See the online help system 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 189.

### RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

#### Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac2004=n
```

### Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

```
-I..\src or -I ..\src\
```

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

### Rules for options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

For example, a short option with an optional parameter followed by a mandatory parameter:

```
-lA MyList.lst
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n PreprocOutput.lst
```

### Rules for specifying a filename or directory as parameters

These rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally take a file path. The path can be relative or absolute. For example, to generate a listing to the file `List.lst` in the directory `..\listings\`:

```
iccr16c 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:

```
icccr16c 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:
- ```
icccr16c 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:

```
icccr16c 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`:

```
icccr16c prog.c -l ---r
```

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

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

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

```
--diag_warning=Be0001
```

```
--diag_warning=Be0002
```

---

## Summary of compiler options

This table summarizes the compiler command line options:

| Command line option             | Description                                        |
|---------------------------------|----------------------------------------------------|
| <code>--c89</code>              | Specifies the C89 dialect                          |
| <code>--char_is_signed</code>   | Treats <code>char</code> as signed                 |
| <code>--char_is_unsigned</code> | Treats <code>char</code> as unsigned               |
| <code>--clib</code>             | Uses the system include files for the CLIB library |
| <code>--code_model</code>       | Specifies the code model                           |

Table 31: Compiler options summary

| Command line option              | Description                                                                                                                                               |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| -D                               | Defines preprocessor symbols                                                                                                                              |
| --data_model                     | Specifies the data model                                                                                                                                  |
| --debug                          | Generates debug information                                                                                                                               |
| --dependencies                   | Lists file dependencies                                                                                                                                   |
| --diag_error                     | Treats these as errors                                                                                                                                    |
| --diag_remark                    | Treats these as remarks                                                                                                                                   |
| --diag_suppress                  | Suppresses these diagnostics                                                                                                                              |
| --diag_warning                   | Treats these as warnings                                                                                                                                  |
| --diagnostics_tables             | Lists all diagnostic messages                                                                                                                             |
| --discard_unused_publics         | Discards unused public symbols                                                                                                                            |
| --dlib                           | Uses the system include files for the DLIB library                                                                                                        |
| --dlib_config                    | Uses the system include files for the DLIB library and determines which configuration of the library to use                                               |
| -e                               | Enables language extensions                                                                                                                               |
| --ec++                           | Specifies Embedded C++                                                                                                                                    |
| --eec++                          | Specifies Extended Embedded C++                                                                                                                           |
| --enable_multibytes              | Enables support for multibyte characters in source files                                                                                                  |
| --error_limit                    | Specifies the allowed number of errors before compilation stops                                                                                           |
| -f                               | Extends the command line                                                                                                                                  |
| --guard_calls                    | Enables guards for function static variable initialization                                                                                                |
| --header_context                 | Lists all referred source files and header files                                                                                                          |
| -I                               | Specifies include file path                                                                                                                               |
| -l                               | Creates a list file                                                                                                                                       |
| --library_module                 | Creates a library module                                                                                                                                  |
| --macro_positions_in_diagnostics | Obtains positions inside macros in diagnostic messages                                                                                                    |
| --mfc                            | Enables multi-file compilation                                                                                                                            |
| --misrac                         | Enables error messages specific to MISRA-C:1998. This option is a synonym of <code>--misrac1998</code> and is only available for backwards compatibility. |

*Table 31: Compiler options summary (Continued)*

| Command line option                       | Description                                                                                                                      |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>--misrac1998</code>                 | Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .           |
| <code>--misrac2004</code>                 | Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .           |
| <code>--misrac_verbose</code>             | <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> or the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> . |
| <code>--module_name</code>                | Sets the object module name                                                                                                      |
| <code>--no_code_motion</code>             | Disables code motion optimization                                                                                                |
| <code>--no_cse</code>                     | Disables common subexpression elimination                                                                                        |
| <code>--no_inline</code>                  | Disables function inlining                                                                                                       |
| <code>--no_path_in_file_macros</code>     | Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>                       |
| <code>--no_size_constraints</code>        | Relaxes the normal restrictions for code size expansion when optimizing for speed.                                               |
| <code>--no_static_destruction</code>      | Disables destruction of C++ static variables at program exit                                                                     |
| <code>--no_system_include</code>          | Disables the automatic search for system include files                                                                           |
| <code>--no_tbaa</code>                    | Disables type-based alias analysis                                                                                               |
| <code>--no_typedefs_in_diagnostics</code> | Disables the use of typedef names in diagnostics                                                                                 |
| <code>--no_unroll</code>                  | Disables loop unrolling                                                                                                          |
| <code>--no_warnings</code>                | Disables all warnings                                                                                                            |
| <code>--no_wrap_diagnostics</code>        | Disables wrapping of diagnostic messages                                                                                         |
| <code>-O</code>                           | Sets the optimization level                                                                                                      |
| <code>-o</code>                           | Sets the object filename. Alias for <code>--output</code> .                                                                      |
| <code>--omit_types</code>                 | Excludes type information                                                                                                        |
| <code>--only_stdout</code>                | Uses standard output only                                                                                                        |
| <code>--output</code>                     | Sets the object filename                                                                                                         |
| <code>--predef_macros</code>              | Lists the predefined symbols.                                                                                                    |
| <code>--preinclude</code>                 | Includes an include file before reading the source file                                                                          |

Table 31: Compiler options summary (Continued)

| Command line option                      | Description                                                   |
|------------------------------------------|---------------------------------------------------------------|
| <code>--preprocess</code>                | Generates preprocessor output                                 |
| <code>--public_equ</code>                | Defines a global named assembler label                        |
| <code>-r</code>                          | Generates debug information. Alias for <code>--debug</code> . |
| <code>--relaxed_fp</code>                | Relaxes the rules for optimizing floating-point expressions   |
| <code>--remarks</code>                   | Enables remarks                                               |
| <code>--require_prototypes</code>        | Verifies that functions are declared before they are defined  |
| <code>--segment</code>                   | Changes a segment name                                        |
| <code>--silent</code>                    | Sets silent operation                                         |
| <code>--strict</code>                    | Checks for strict compliance with Standard C/C++              |
| <code>--system_include_dir</code>        | Specifies the path for system include files                   |
| <code>--use_c++_inline</code>            | Uses C++ inline semantics in C99                              |
| <code>--vla</code>                       | Enables C99 VLA support                                       |
| <code>--warnings_affect_exit_code</code> | Warnings affect exit code                                     |
| <code>--warnings_are_errors</code>       | Warnings are treated as errors                                |

Table 31: Compiler options summary (Continued)

## Descriptions of compiler options

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



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

### **--c89**

Syntax

`--c89`

Description

Use this option to enable the C89 C dialect instead of Standard C.

**Note:** This option is mandatory when the MISRA C checking is enabled.

See also

*C language overview*, page 145.



**Project>Options>C/C++ Compiler>Language 1>C dialect>C89**

## --char\_is\_signed

Syntax

`--char_is_signed`

Description

By default, the compiler interprets the plain `char` type as unsigned. Use this option to make the compiler interpret the plain `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

**Note:** The runtime library is compiled without the `--char_is_signed` option and cannot be used with code that is compiled with this option.

**Project>Options>C/C++ Compiler>Language 2>Plain ‘char’ is**

## --char\_is\_unsigned

Syntax

`--char_is_unsigned`

Description

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

**Project>Options>C/C++ Compiler>Language 2>Plain ‘char’ is**

## --clib

Syntax

`--clib`

Description

Use this option to use the system header files for the CLIB library; the compiler will automatically locate the files and use them when compiling.

**Note:** By default, the compiler uses the DLIB library.

See also

`--dlib`, page 207 and `--dlib_config`, page 207.



To set related options, choose:

**Project>Options>General Options>Library Configuration**

## --code\_model

|             |                                                                                                                                                                                                                                     |                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| Syntax      | <code>--code_model={normal short}</code>                                                                                                                                                                                            |                                   |
| Parameters  | <code>normal</code> (default)                                                                                                                                                                                                       | Supports the Normal register mode |
|             | <code>short</code>                                                                                                                                                                                                                  | Supports the Short register mode  |
| Description | Use this option to select the code model, which means a default register mode. If you do not select a code model, the compiler uses the default code model. Note that all modules of your application must use the same code model. |                                   |
| See also    | <i>Code models and register modes</i> , page 55.                                                                                                                                                                                    |                                   |



**Project>Options>General Options>Target>Code model**

## -D

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|
| Syntax      | <code>-D symbol[=value]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                      |
| Parameters  | <code>symbol</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | The name of the preprocessor symbol  |
|             | <code>value</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | The value of the preprocessor symbol |
| Description | <p>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.</p> <p>The option <code>-D</code> has the same effect as a <code>#define</code> statement at the top of the source file:</p> <pre>-Dsymbol</pre> <p>is equivalent to:</p> <pre>#define symbol 1</pre> <p>To get the equivalence of:</p> <pre>#define FOO</pre> <p>specify the = sign but nothing after, for example:</p> <pre>-DFOO=</pre> |                                      |



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

## --data\_model

|             |                                                                                                                                                                                                                                                 |                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------|
| Syntax      | <code>--data_model={small medium large huge indexed}</code>                                                                                                                                                                                     |                                                   |
| Parameters  | <code>small</code> (default)                                                                                                                                                                                                                    | Places data objects in data16 memory by default.  |
|             | <code>medium</code>                                                                                                                                                                                                                             | Places data objects in data20 memory by default.  |
|             | <code>large</code>                                                                                                                                                                                                                              | Places data objects in data24 memory by default.  |
|             | <code>huge</code>                                                                                                                                                                                                                               | Places data objects in data32 memory by default.  |
|             | <code>indexed</code>                                                                                                                                                                                                                            | Places data objects in index20 memory by default. |
| Description | Use this option to select the data model, which means a default placement of data objects. If you do not select a data model, the compiler uses the default data model. Note that all modules of your application must use the same data model. |                                                   |
| See also    | <i>Data models</i> , page 44.                                                                                                                                                                                                                   |                                                   |



**Project>Options>General Options>Target>Data model**

## --debug, -r

|             |                                                                                                                                                                                         |  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--debug</code><br><code>-r</code>                                                                                                                                                 |  |
| Description | Use the <code>--debug</code> or <code>-r</code> option to make the compiler include information in the object modules required by the IAR C-SPY® Debugger and other symbolic debuggers. |  |
|             | <b>Note:</b> Including debug information will make the object files larger than otherwise.                                                                                              |  |



**Project>Options>C/C++ Compiler>Output>Generate debug information**

## --dependencies

|            |                                                            |                               |
|------------|------------------------------------------------------------|-------------------------------|
| Syntax     | <code>--dependencies [= [i m]] {filename directory}</code> |                               |
| Parameters | <code>i</code> (default)                                   | Lists only the names of files |
|            | <code>m</code>                                             | Lists in makefile style       |

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 196.

**Description** Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension `i`.

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

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

If `--dependencies=m` is used, the output is in makefile style. For each input file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of an input file. For example:

```
foo.r45: c:\iar\product\include\stdio.h
foo.r45: d:\myproject\include\foo.h
```

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

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

```
%.r45 : %.c
        $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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



This option is not available in the IDE.

## **--diag\_error**

**Syntax** `--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      | <code>--diag_warning=tag[, tag, ...]</code>                                                                                                                                                                                                                                    |                                                                                       |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                     | The number of a diagnostic message, for example the message number <code>Pe826</code> |
| Description | Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line. |                                                                                       |



**Project>Options>C/C++ CompilerLinker>Diagnostics>Treat these as warnings**

## --diagnostics\_tables

|             |                                                                                                                                                                                                                                                     |  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--diagnostics_tables {filename directory}</code>                                                                                                                                                                                              |  |
| Parameters  | For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 196.                                                                                                       |  |
| 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      | <code>--discard_unused_publics</code>                                                                                                  |  |
| Description | Use this option to discard unused public functions and variables when compiling with the <code>--mfc</code> compiler option.           |  |
|             | <b>Note:</b> Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output. |  |
| See also    | <code>--mfc</code> , page 213 and <i>Multi-file compilation units</i> , page 177.                                                      |  |

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

|             |                                                                                                                                                                                                                                                                 |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --dlib                                                                                                                                                                                                                                                          |
| Description | Use this option to use the system header files for the DLIB library; the compiler will automatically locate the files and use them when compiling.<br><b>Note:</b> The DLIB library is used by default: To use the CLIB library, use the --clib option instead. |
| See also    | --dlib_config, page 207, --no_system_include, page 215, --system_include_dir, page 224, and --clib, page 201.                                                                                                                                                   |



To set related options, choose:

**Project>Options>General Options>Library Configuration****--dlib\_config**

|             |                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --dlib_config <i>filename.h</i>   <i>config</i>                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                |
| Parameters  | <i>filename</i>                                                                                                                                                                                                                                                                                                                                                                                             | A DLIB configuration header file. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 196.                                                                                               |
|             | <i>config</i>                                                                                                                                                                                                                                                                                                                                                                                               | The default configuration file for the specified configuration will be used. Choose between:<br><br>none, no configuration will be used<br><br>normal, the normal library configuration will be used<br><br>full, the full library configuration will be used. |
| Description | Use this option to specify which library configuration to use, either by specifying an explicit file or by specifying a library configuration—in which case the default file for that library configuration will be used. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used. |                                                                                                                                                                                                                                                                |

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `cr16c\lib`. For examples and information about prebuilt runtime libraries, see *Using a prebuilt library*, page 81.

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 91.

**Note:** This option only applies to the IAR DLIB runtime environment.



To set related options, choose:

**Project>Options>General Options>Library Configuration**

## **-e**

Syntax

`-e`

Description

In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.

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

See also

*Enabling language extensions, page 147.*



**Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions**

**Note:** By default, this option is selected in the IDE.

## **--ec++**

Syntax

`--ec++`

Description

In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.




**Project>Options>C/C++ Compiler>Language 1>C++**

and


**Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++**



**--eec++**

|             |                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--eec++</code>                                                                                                                                                                                                                                               |
| Description | In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++. |
| See also    | <i>Extended Embedded C++</i> , page 154.                                                                                                                                                                                                                           |
|             |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 1&gt;C++</b><br>and<br><b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 1&gt;C++ dialect&gt;Extended Embedded C++</b>     |

**--enable\_multibytes**

|             |                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--enable_multibytes</code>                                                                                                                                                                                                                                                                                                                                                                           |
| Description | By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.<br><br>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. |
|             |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 2&gt;Enable multibyte support</b>                                                                                                                                                                                                                                  |

**--error\_limit**

|             |                                                                                                                                                                                   |          |                                                                                                                            |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--error_limit=<i>n</i></code>                                                                                                                                               |          |                                                                                                                            |
| Parameters  | <table> <tr> <td><i>n</i></td> <td>The number of errors before the compiler stops the compilation. <i>n</i> must be a positive integer; 0 indicates no limit.</td> </tr> </table> | <i>n</i> | The number of errors before the compiler stops the compilation. <i>n</i> must be a positive integer; 0 indicates no limit. |
| <i>n</i>    | The number of errors before the compiler stops the compilation. <i>n</i> must be a positive integer; 0 indicates no limit.                                                        |          |                                                                                                                            |
| Description | Use the <code>--error_limit</code> 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      | <code>-f filename</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 196.                                                                                                                                                                                                                                                                                                                                                                                               |
| Description | <p>Use this option to make the compiler read command line options from the named file, with the default filename extension <code>.xcl</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.</p> <p>Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.</p> |



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

## **--guard\_calls**

|             |                                                                                                                                                                                                                                                                                 |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--guard_calls</code>                                                                                                                                                                                                                                                      |
| Description | <p>Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.</p> <p><b>Note:</b> This option requires a threaded C++ environment, which is not supported in the IAR C/C++ Compiler for CR16C.</p> |



This option is not available in the IDE.

## **--header\_context**

|             |                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--header_context</code>                                                                                                                                                 |
| 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      | <code>-I path</code>                                                                                                                     |
| Parameters  | <code>path</code> The search path for <code>#include</code> files                                                                        |
| Description | Use this option to specify the search paths for <code>#include</code> files. This option can be used more than once on the command line. |
| See also    | <i>Include file search procedure</i> , page 190.                                                                                         |



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

## -l

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|---------------------|---|------------------------------------------------------|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|--------------------|-------------|------------------------------------------------------|
| Syntax      | <code>-l [a A b B c C D] [N] [H] {filename directory}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |
| Parameters  | <table> <tr> <td>a (default)</td> <td>Assembler list file</td> </tr> <tr> <td>A</td> <td>Assembler list file with C or C++ source as comments</td> </tr> <tr> <td>b</td> <td>Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code>, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *</td> </tr> <tr> <td>B</td> <td>Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code>, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *</td> </tr> <tr> <td>c</td> <td>C or C++ list file</td> </tr> <tr> <td>C (default)</td> <td>C or C++ list file with assembler source as comments</td> </tr> </table> | a (default) | Assembler list file | A | Assembler list file with C or C++ source as comments | b | Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included * | B | Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included * | c | C or C++ list file | C (default) | C or C++ list file with assembler source as comments |
| a (default) | Assembler list file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |
| A           | Assembler list file with C or C++ source as comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |
| b           | Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |
| B           | Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |
| c           | C or C++ list file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |
| C (default) | C or C++ list file with assembler source as comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |             |                     |   |                                                      |   |                                                                                                                                                                                                                                                         |   |                                                                                                                                                                                                                                                         |   |                    |             |                                                      |

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

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

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 196.

**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**

## --library\_module

**Syntax** --library\_module

**Description** Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.



**Project>Options>C/C++ Compiler>Output>Module type>Library Module**

## --macro\_positions\_in\_diagnostics

**Syntax** --macro\_positions\_in\_diagnostics

**Description** Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.



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

## --mfc

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--mfc</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Description | Use this option to enable <i>multi-file compilation</i> . This means that the compiler compiles one or several source files specified on the command line as one unit, which enhances interprocedural optimizations.<br><br><b>Note:</b> The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the <code>-o</code> compiler option and specify a certain output file. |
| Example     | <code>icccr16c myfile1.c myfile2.c myfile3.c --mfc</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| See also    | <code>--discard_unused_publics</code> , page 206, <code>--output</code> , <code>-o</code> , page 218, and <i>Multi-file compilation units</i> , page 177.                                                                                                                                                                                                                                                                                                                                                                       |



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

## --module\_name

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--module_name=<i>name</i></code>                                                                                                                                                                                                                                                                                                                                                                                             |
| Parameters  | <i>name</i> An explicit object module name                                                                                                                                                                                                                                                                                                                                                                                         |
| Description | Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.<br><br>This option is useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor. |



**Project>Options>C/C++ Compiler>Output>Object module name**

## --no\_code\_motion

|             |                                                       |
|-------------|-------------------------------------------------------|
| Syntax      | <code>--no_code_motion</code>                         |
| Description | Use this option to disable code motion optimizations. |

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

See also

*Code motion*, page 180.



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

## **--no\_cse**

Syntax

`--no_cse`

Description

Use this option to disable common subexpression elimination.

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

See also

*Common subexpression elimination*, page 179.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination**

## **--no\_inline**

Syntax

`--no_inline`

Description

Use this option to disable function inlining.

See also

*Inlining functions*, page 63.



**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__`.

See also

*Description of predefined preprocessor symbols*, page 278.



This option is not available in the IDE.

## --no\_size\_constraints

Syntax `--no_size_constraints`

Description Use this option to relax the normal restrictions for code size expansion when optimizing for high speed.

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

See also *Speed versus size*, page 179.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>No size constraints**

## --no\_static\_destruction

Syntax `--no_static_destruction`

Description Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.

Use this option to suppress the emission of such code.



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

## --no\_system\_include

Syntax `--no_system_include`

Description By default, the compiler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the `-I` compiler option.

See also *--dlib*, page 207, *--dlib\_config*, page 207, and *--system\_include\_dir*, page 224.



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

## --no\_tbaa

Syntax `--no_tbaa`

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

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

See also *Type-based alias analysis*, page 180.



**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\_unroll

Syntax `--no_unroll`

Description Use this option to disable loop unrolling.

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



See also *Loop unrolling*, page 180.



**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|1|m|h|hs|hz]`

Parameters

|             |                            |
|-------------|----------------------------|
| n           | None* (Best debug support) |
| 1 (default) | Low*                       |
| m           | Medium                     |
| h           | High, balanced             |
| hs          | High, favoring speed       |
| hz          | High, favoring size        |

\*The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.

**Description** Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only `-O` is used without any parameter, the optimization level High balanced is used.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

**See also** *Controlling compiler optimizations*, page 177.



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

## --omit\_types

**Syntax** `--omit_types`

**Description** By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.



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

## --only\_stdout

**Syntax** `--only_stdout`


**Description** Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).




This option is not available in the IDE.

## --output, -o


**Syntax** `--output {filename|directory}`  
`-o {filename|directory}`

|             |                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Parameters  | For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 196.                                                                                                                  |
| 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 <code>.o</code> . 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      | <code>--predef_macros {filename directory}</code>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 196.                                                                                                                                                                                                                                                                                                                                      |
| 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.<br><br>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 <code>predef</code> filename extension.<br><br>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      | <code>--preinclude includefile</code>                                                                                                                                                                                                                         |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 196.                                                                                                                                |
| Description | Use this option to make the compiler read the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol. |
|             |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Preprocessor&gt;Preinclude file</b>                                                                                            |

## --preprocess

Syntax `--preprocess [= [c] [n] [l]] {filename|directory}`

### Parameters

|   |                           |
|---|---------------------------|
| c | Preserve comments         |
| n | Preprocess only           |
| l | Generate #line directives |

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 196.

### Description

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



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

## --public\_equ

Syntax `--public_equ symbol [=value]`

### Parameters

|               |                                                   |
|---------------|---------------------------------------------------|
| <i>symbol</i> | The name of the assembler symbol to be defined    |
| <i>value</i>  | An optional value of the defined assembler symbol |

### Description

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.

## --relaxed\_fp

Syntax `--relaxed_fp`

### Description

Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:

- The expression consists of both single- and double-precision values

- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

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

### Example

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

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



To set related options, choose:

**Project>Options>C/C++ Compiler>Language 2>Floating-point semantics**

## --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 193.



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

## --require\_prototypes

### Syntax

```
--require_prototypes
```

### Description

Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration

- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.



**Project>Options>C/C++ Compiler>Language 1>Require prototypes**

## --segment

### Syntax

```
--segment {memattr|segmentname}=NEWSEGMENTNAME
```

### Parameters

*memattr*            A data memory attribute. All data objects declared with this memory attribute are placed in segments with names that begin with *NEWSEGMENTNAME*.

*segmentname*        One of these segment names:

```
absolute_c
absolute_n
code
cstack
difunct
icode
idinit
intvec
istack
trapvec
zinit
```

The name of the segment *segmentname* changes to *NEWSEGMENTNAME*.

### Description

The compiler places functions and data objects into named segments which are referred to by the IAR XLINK Linker. Use the `--segment` option to place functions and/or data objects in named segments.

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 segment names require corresponding modifications in the linker configuration file.

**Note:** To place `__ix4` scalars larger than a character and structures without `char` members (in other words, variable data that by default is placed in segments with the base name `INDEX5`) in a renamed segment, use the *memattr* parameter `__ix5`.

**Example**

This command places the `__data20 int a;` defined variable in the `MYDATA_Z` segment:

```
--segment __data20=MYDATA
```

This command renames the segment that contains interrupt vectors to `MYINTS` instead of the default name `INTVEC`:

```
--segment intvec=MYINTS
```

**See also**

*Controlling data and function placement in memory*, page 173 and *Summary of extended keywords*, page 242.



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****Syntax**

```
--strict
```

**Description**

By default, the compiler accepts a relaxed superset of Standard C and C++. Use this option to ensure that the source code of your application instead conforms to strict Standard C and C++.

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


**See also**

*Enabling language extensions*, page 147.




**Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict**

## --system\_include\_dir

|             |                                                                                                                                                                                                                                                             |                                                                                                                                                                  |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--system_include_dir path</code>                                                                                                                                                                                                                      |                                                                                                                                                                  |
| Parameters  | <i>path</i>                                                                                                                                                                                                                                                 | The path to the system include files. For information about specifying a path, see <i>Rules for specifying a filename or directory as parameters</i> , page 196. |
| Description | By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location. |                                                                                                                                                                  |
| See also    | <code>--dlib</code> , page 207, <code>--dlib_config</code> , page 207, and <code>--no_system_include</code> , page 215.                                                                                                                                     |                                                                                                                                                                  |
|             |                                                                                                                                                                            | This option is not available in the IDE.                                                                                                                         |

## --use\_c++\_inline

|             |                                                                                                                                                                 |                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--use_c++_inline</code>                                                                                                                                   |                                                                                                       |
| Description | Standard C uses slightly different semantics for the <code>inline</code> keyword than C++ does. Use this option if you want C++ semantics when you are using C. |                                                                                                       |
| See also    | <i>Inlining functions</i> , page 63                                                                                                                             |                                                                                                       |
|             |                                                                              | <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 1&gt;C dialect&gt;C99&gt;C++ inline semantics</b> |

## --vla

|             |                                                                                                                                                                                                                 |  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--vla</code>                                                                                                                                                                                              |  |
| Description | Use this option to enable support for C99 variable length arrays. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the <code>--c89</code> compiler option. |  |
|             | <b>Note:</b> <code>--vla</code> should not be used together with the <code>longjmp</code> library function, as that can lead to memory leakages.                                                                |  |
| See also    | <i>C language overview</i> , page 145.                                                                                                                                                                          |  |



**Project>Options>C/C++ Compiler>Language 1>C dialect>Allow VLA**

## **--warnings\_affect\_exit\_code**

|             |                                                                                                                                                                              |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--warnings_affect_exit_code</code>                                                                                                                                     |
| Description | By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code. |



This option is not available in the IDE.

## **--warnings\_are\_errors**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--warnings_are_errors</code>                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 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.<br><br><b>Note:</b> Any diagnostic messages that have been reclassified as warnings by the option <code>--diag_warning</code> or the <code>#pragma diag_warning</code> directive will also be treated as errors when <code>--warnings_are_errors</code> is used. |
| See also    | <code>--diag_warning</code> , page 206.                                                                                                                                                                                                                                                                                                                                                                                                                                       |

**Project>Options>C/C++ Compiler>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 that 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`.

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 235.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

### ALIGNMENT ON THE CR16C MICROPROCESSOR

The CR16C microprocessor can access aligned objects more efficiently than non-aligned objects. Objects with alignment 2 must be stored at addresses divisible by 2.

## Basic data types

The compiler supports both all Standard C basic data types and some additional types.

### INTEGER TYPES

This table gives the size and range of each integer data type:

| Data type                       | Size    | Range                   | Alignment |
|---------------------------------|---------|-------------------------|-----------|
| <code>bool</code>               | 8 bits  | 0 to 1                  | 1         |
| <code>char</code>               | 8 bits  | 0 to 255                | 1         |
| <code>signed char</code>        | 8 bits  | -128 to 127             | 1         |
| <code>unsigned char</code>      | 8 bits  | 0 to 255                | 1         |
| <code>signed short</code>       | 16 bits | -32768 to 32767         | 2         |
| <code>unsigned short</code>     | 16 bits | 0 to 65535              | 2         |
| <code>signed int</code>         | 16 bits | -32768 to 32767         | 2         |
| <code>unsigned int</code>       | 16 bits | 0 to 65535              | 2         |
| <code>signed long</code>        | 32 bits | $-2^{31}$ to $2^{31}-1$ | 2         |
| <code>unsigned long</code>      | 32 bits | 0 to $2^{32}-1$         | 2         |
| <code>signed long long</code>   | 32 bits | $-2^{31}$ to $2^{31}-1$ | 2         |
| <code>unsigned long long</code> | 32 bits | 0 to $2^{32}-1$         | 2         |

Table 32: Integer types

Signed variables are represented using the two's complement form.

### Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

### The long long type

The `long long` data type is supported with these restrictions:

- The CLIB runtime library does not support the `long long` type
- There is no support for the 64-bit `long long` type. This is formally a violation of the C standard.

### The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example:

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

### The `char` type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

### The `wchar_t` type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

**Note:** The IAR CLIB Library has only rudimentary support for `wchar_t`.

### Bitfields

In Standard C, `int`, `signed int`, and `unsigned int` can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (`char`, `short`, `int`, etc) results in a signed or unsigned bitfield.

In the IAR C/C++ Compiler for CR16C, plain integer types in bitfields are treated as unsigned.

Bitfields in expressions are treated as `int` if `int` can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.

Each bitfield is placed into a container of its base type from the least significant bit to the most significant bit. If the last container is of the same type and has enough bits available, the bitfield is placed into this container, otherwise a new container is allocated.

If you use the directive `#pragma bitfield=reversed`, bitfields are placed from the most significant bit to the least significant bit in each container. See *bitfields*, page 253.

### **Example**

Assume this example:

```
struct BitfieldExample
{
    uint32_t a : 12;
    uint16_t b : 3;
    uint16_t c : 7;
    uint8_t  d;
};
```

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset 0 and puts `a` into the least significant 12 bits of the container.

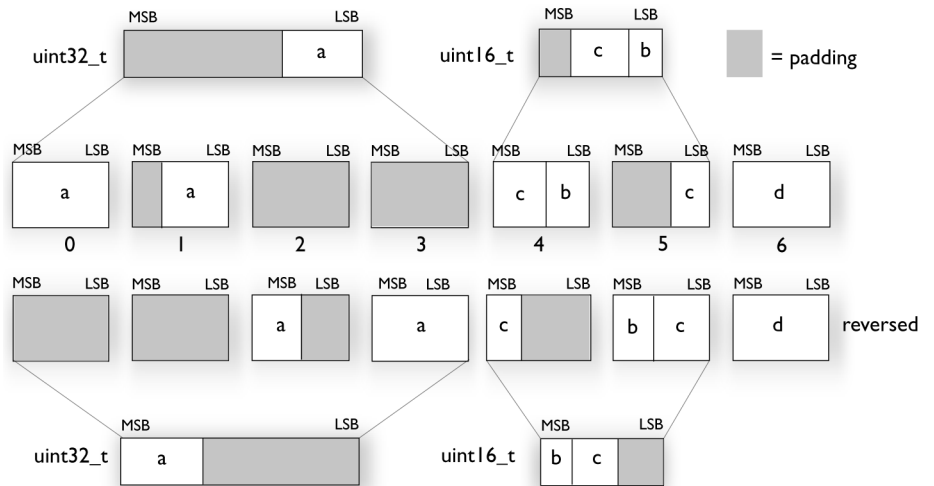
To place the second bitfield, `b`, a new container is allocated at offset 4, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

The fourth member, `d`, is allocated into the byte at offset 6. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order, each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example`:



## FLOATING-POINT TYPES

In the IAR C/C++ Compiler for CR16C, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

| Type        | Size    |
|-------------|---------|
| float       | 32 bits |
| double      | 64 bits |
| long double | 64 bits |

Table 33: Floating-point types

## Floating-point environment

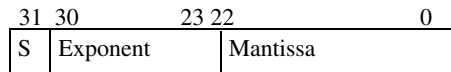
Exception flags are not supported. The `feraiseexcept` function does not raise any exceptions.

Exception flags for floating-point values are supported, and they are defined in the `feenv.h` file.

The `feraiseexcept` function raises an `inexact` floating-point exception when called with `FE_OVERFLOW` or `FE_UNDERFLOW`. The `feraiseexcept` function does not raise an `inexact` floating-point exception when called with `FE_OVERFLOW` or `FE_UNDERFLOW`.

### 32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:



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

The value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-127)} * 1.\text{Mantissa}$$

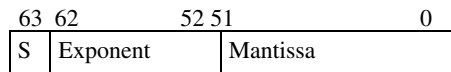
The range of the number is at least:

$$\pm 1.18\text{E}-38 \text{ to } \pm 3.39\text{E}+38$$

The precision of the float operators (+, -, \*, and /) is approximately 7 decimal digits.

### 64-bit floating-point format

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



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-1023)} * 1.\text{Mantissa}$$

The range of the number is at least:

$$\pm 2.23\text{E}-308 \text{ to } \pm 1.79\text{E}+308$$

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 the most significant bit in the mantissa to 1. The value of the sign bit is ignored.



- Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is subnormal, even though the number is treated as if the exponent was 1. Unlike normal numbers, subnormal numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a subnormal number is:

$$(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$$

where `BIAS` is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

The smallest positive and negative numbers representable by denormalized numbers are:

$\pm 1.40E-45$  (for a 32-bit floating-point number)

$\pm 4.94E-324$  (for a 64-bit double)

**Note:** The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, `NaN`, and subnormal numbers. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

---

## Pointer types

The compiler has two basic types of pointers: code pointers and data pointers.

### SIZE

The size of code pointers is always 32 bits and they can address the entire memory. The internal representation of a code pointer is the actual address it refers to divided by two.

Data pointers have two sizes: 16 or 32 bits. The former is used for data16 memory and the latter for all other memory types. The index type of 16-bit pointers is `int` and of 32-bit pointers it is `long`.

### 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 integer type to a pointer of a larger type is performed by zero 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
- Casting from a smaller pointer to a larger pointer is performed by zero extension.
- Casting from a larger pointer to a smaller pointer is performed by truncation.

### **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 CR16C, the size of `size_t` is 32 bits.

Note that for the Small data model, this is formally a violation of the standard; the size of `size_t` should actually be 16 bits.

### **ptrdiff\_t**

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the IAR C/C++ Compiler for CR16C, the size of `ptrdiff_t` is 32 bits.

For the Small data model, this is formally a violation of the standard; the size of `ptrdiff_t` should actually be 16 bits.

### **intptr\_t**

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for CR16C, the size of `intptr_t` is 32 bits.

### **uintptr\_t**

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned.

## **ALIGNMENT**

The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. The size of a `struct` is also adjusted to allow arrays of aligned structure objects.

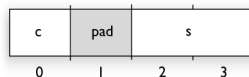
## **GENERAL LAYOUT**

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

### Example

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

This diagram shows the layout in memory:



The alignment of the structure is 2 bytes, and a pad byte must be inserted to give `short s` the correct alignment.

### PACKED STRUCTURE TYPES

The `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work, because the normal code might depend on the alignment being correct.

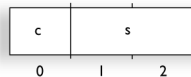
### Example

This example declares a packed structure:

```
#pragma pack(1)
struct S
{
    char c;
    short s;
};

#pragma pack()
```

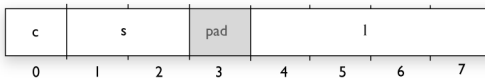
In this example, the structure `s` has this memory layout:



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

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

`S2` has this memory layout



The structure `s` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 2, which means that alignment of the structure `S2` will become 2.

For more information, see *Alignment of elements in a structure*, page 171.

---

## Type qualifiers

According to the C standard, `volatile` and `const` are type qualifiers.

### DECLARING OBJECTS VOLATILE

By declaring an object `volatile`, the compiler is informed that the value of the object can change beyond the compiler's control. The compiler must also assume that any accesses can have side effects—thus all accesses to the `volatile` object must be preserved.

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

## Definition of access to volatile objects

The C standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine:

- The compiler considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type
- Adding a `const` qualifier to a `volatile` object will make write accesses to the object impossible. However, the object will be placed in RAM as specified by the C standard.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for CR16C are described below.

## Rules for accesses

In the IAR C/C++ Compiler for CR16C, 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 8-bit and 16-bit scalars.

For all combinations of object types not listed, only the rule that states that all accesses are preserved applies.

## DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, follow this example:

```
/* Header */
extern int const xVar;
#define x (*(int const volatile *) &xVar)

/* Source that uses x */
int DoSomething()
{
    return x;
}

/* Source that defines x */
#pragma segment = "FLASH"
int const xVar @ "FLASH" = 6;
```

The segment `FLASH` contains the initializers. They must be flashed manually when the application starts up.

Thereafter, the initializers can be reflashed with other values at any time.

## DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` are allocated in ROM, except if they are located in `index4` or `index5` memory—in that case they are allocated in RAM.

In C++, objects that require runtime initialization cannot be placed in ROM.

---

## Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.

# Extended keywords

This chapter describes the extended keywords that support specific features of the CR16C microprocessor and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

For information about the address ranges of the different memory areas, see the chapter *Segment reference*.

---

## 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 CR16C microprocessor. 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 information about how to use attributes to modify data, see the chapter *Data storage*. For more information about each attribute, see *Descriptions of extended keywords*, page 243.

**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 208 for more information.

### TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes explicitly in your declarations, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

### Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microprocessor.

- Available *data memory attributes*: `__data16`, `__data20`, `__data24`, `__data32`, `__ix4`, and `__ix20`.

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is implicitly used by the compiler. You can specify one memory attribute for each level of pointer indirection.

### General type attributes

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt`, `__monitor`, `__task`, `__trap`, and `__user`

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

### Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the `__data20` type attribute to the variables `i` and `j`; in other words, the variables `i` and `j` are placed in data20 memory. However, note that an individual member of a `struct` or `union` cannot have a type attribute. The variables `k` and `l` behave in the same way:

```
__data20 int i, j;
int __data20 k, l;
```

Note that the attribute affects both identifiers.

This declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__data20
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. Note that the pragma directive has no effect if a memory attribute is already explicitly declared.

For more examples of using memory attributes, see *More examples*, page 50.

An easier way of specifying storage is to use type definitions. These two declarations are equivalent:

```
typedef char __data20 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__data20 char b;
char __data20 *bp;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

### Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

|                                |                                                                                                          |
|--------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>int __data20 * p;</code> | The <code>int</code> object is located in <code>data20</code> memory.                                    |
| <code>int * __data20 p;</code> | The pointer is located in <code>data20</code> memory.                                                    |
| <code>__data20 int * p;</code> | If you declare a variable, the pointer is located in <code>data20</code> memory.                         |
|                                | If you declare a structure member, the <code>int</code> object is located in <code>data20</code> memory. |

### 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:

```
__interrupt void my_handler(void);
or
void (__interrupt my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

## OBJECT ATTRIBUTES

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__noreturn`, `__noadjust`, `__raw`, and `vector`.

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 173. For more information about `vector`, see *vector*, page 266.

### Syntax for object attributes

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

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

The `#pragma object_attribute` directive can also be used. This declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

**Note:** Object attributes cannot be used in combination with the `typedef` keyword.

---

## Summary of extended keywords

This table summarizes the extended keywords:

| Extended keyword      | Description                          |
|-----------------------|--------------------------------------|
| <code>__data16</code> | Controls the storage of data objects |

Table 34: Extended keywords summary

| Extended keyword         | Description                                                                       |
|--------------------------|-----------------------------------------------------------------------------------|
| <code>__data20</code>    | Controls the storage of data objects                                              |
| <code>__data24</code>    | Controls the storage of data objects                                              |
| <code>__data32</code>    | Controls the storage of data objects                                              |
| <code>__interrupt</code> | Specifies interrupt functions                                                     |
| <code>__intrinsic</code> | Reserved for compiler internal use only                                           |
| <code>__ix4</code>       | Controls the storage of data objects                                              |
| <code>__ix20</code>      | Controls the storage of data objects                                              |
| <code>__monitor</code>   | Specifies atomic execution of a function                                          |
| <code>__near</code>      | Alias for <code>__data16</code> , available for backward compatibility            |
| <code>__noadjust</code>  | Inhibits the default adjustment of the return address of trap functions           |
| <code>__no_init</code>   | Places a data object in non-volatile memory                                       |
| <code>__noreturn</code>  | Informs the compiler that the function will not return                            |
| <code>__raw</code>       | Prevents saving used registers in interrupt functions                             |
| <code>__root</code>      | Ensures that a function or variable is included in the object code even if unused |
| <code>__sbrel</code>     | Alias for <code>__ix4</code> , available for backward compatibility               |
| <code>__task</code>      | Relaxes the rules for preserving registers                                        |
| <code>__trap</code>      | Supports trap functions                                                           |
| <code>__user</code>      | Specifies the <code>JUSR</code> instruction for function calls                    |

Table 34: Extended keywords summary (Continued)

## Descriptions of extended keywords

These sections give detailed information about each extended keyword.

### `__data16`

#### Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

#### Description

The `__data16` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data16 memory.

You can also use the `__data16` attribute to create a pointer explicitly pointing to an object located in the data16 memory.

- Storage information
- Address range: 0–0xFFFF
  - Maximum object size: 32 Kbytes
  - Pointer size: 2 bytes

Example `__data16 int x;`

See also *Memory types*, page 46.

## **\_\_data20**

Syntax Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description The `__data20` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data20 memory. You can also use the `__data20` attribute to create a pointer explicitly pointing to an object located in the data20 memory.

- Storage information
- Address range: 0–0xFFFFF
  - Maximum object size: limited by the amount of available memory
  - Pointer size: 4 bytes

Example `__data20 int x;`

See also *Memory types*, page 46.

## **\_\_data24**

Syntax Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description The `__data24` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data24 memory. You can also use the `__data24` attribute to create a pointer explicitly pointing to an object located in the data24 memory.

- Storage information
- Address range: 0–0FFFFFF
  - Maximum object size: limited by the amount of available memory
  - Pointer size: 4 bytes

Example `__data24 int x;`

See also *Memory types*, page 46.

## **\_\_data32**

Syntax Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description The `__data32` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data32 memory. You can also use the `__data32` attribute to create a pointer explicitly pointing to an object located in the data32 memory.

Storage information

- Address range: 0-0xFFFFFFFF
- Maximum object size: limited by the amount of available memory
- Pointer size: 4 bytes

Example `__data32 int x;`

See also *Memory types*, page 46.

## **\_\_interrupt**

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description The `__interrupt` keyword specifies interrupt functions. To specify one or several interrupt vectors, use the `#pragma vector` directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a `void` return type and cannot have any parameters.

The header file `iodevice.h`, where *device* corresponds to the selected device, contains predefined names for the existing interrupt vectors.

Example 

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

See also *Interrupt functions*, page 56, *vector*, page 266, and *INTVEC*, page 309.

## **\_\_intrinsic**

**Description** The `__intrinsic` keyword is reserved for compiler internal use only.

## **\_\_ix4**

**Syntax** Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

**Description** The `__ix4` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in `index4` or `index5` memory. See *Indexed addressing*, page 47.

You can also use the `__ix4` attribute to create a pointer explicitly pointing to an object located in `index4` or `index5` memory.

**Storage information**

- Address range: 28 bytes anywhere in memory
- Maximum object size: Up to 28 bytes
- Pointer size: 4 bytes

**Example** `__ix4 int x;`

**See also** *Memory types*, page 46.

## **\_\_ix20**

**Syntax** Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

**Description** The `__ix20` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in `index20` memory.

You can also use the `__ix20` attribute to create a pointer explicitly pointing to an object located in the `index20` memory.

**Storage information**

- Address range: 1 Mbyte anywhere in memory
- Maximum object size: Up to 1 Mbyte, limited by the amount of available memory
- Pointer size: 4 bytes

**Example** `__ix20 int x;`

See also *Memory types*, page 46.

## **\_\_monitor**

- Syntax** Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.
- Description** The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.
- Example**

```
__monitor int get_lock(void);
```
- See also** *Monitor functions*, page 58. For information about related intrinsic functions, see *\_\_disable\_interrupt*, page 271, *\_\_enable\_interrupt*, page 271, *\_\_get\_interrupt\_state*, page 271, and *\_\_set\_interrupt\_state*, page 274, respectively.

## **\_\_noadjust**

- Syntax** Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.
- Description** The `__noadjust` keyword inhibits the default adjustment of the return address of trap functions.
- Note:** This keyword is useful only in conjunction with trap functions.
- Example**

```
#pragma vector=0x14
__noadjust __trap int myTrapFunction(void)
```

## **\_\_no\_init**

- Syntax** Follows the generic syntax rules for object attributes, see *Object attributes*, page 242.
- 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];
```

## **\_\_noreturn**

|             |                                                                                                                                                                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 242.                                                                                                                                                                                                       |
| Description | The <code>__noreturn</code> keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are <code>abort</code> and <code>exit</code> . |
| Example     | <pre>__noreturn void terminate(void);</pre>                                                                                                                                                                                                                                                            |

## **\_\_raw**

|             |                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 242.                                                                                                                                                                                                                                                                                                             |
| Description | This keyword prevents saving used registers in interrupt functions.<br><br>Interrupt functions preserve the contents of all used processor registers at function entrance and restore them at exit. However, for some very special applications, it can be desirable to prevent the registers from being saved at function entrance. This can be accomplished by the use of the keyword <code>__raw</code> . |
| Example     | <pre>__raw __interrupt void my_interrupt_function()</pre>                                                                                                                                                                                                                                                                                                                                                    |

## **\_\_root**

|             |                                                                                                                                                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 242.                                                                                                                                                              |
| Description | A function or variable with the <code>__root</code> 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     | <pre>__root int myarray[10];</pre>                                                                                                                                                                                                                            |
| See also    | For more information about modules, segments, and the link process, see the <i>IAR Linker and Library Tools Reference Guide</i> .                                                                                                                             |

## **\_\_task**

|        |                                                                                                                            |
|--------|----------------------------------------------------------------------------------------------------------------------------|
| Syntax | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 239. |
|--------|----------------------------------------------------------------------------------------------------------------------------|



**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.

Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared `__task`.

**Example**

```
__task void my_handler(void);
```

## **\_\_trap**

**Syntax** Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

**Description** A trap function is called and then executed by the `EXCP` assembler instruction and returned by the `RETX` instruction. To specify one or several vectors, use the `#pragma vector` directive. See the chip manufacturer's hardware documentation for information about the trap vector range. If a trap vector is not given, an error will be issued if the function is called. A trap function can take parameters and return a value and it has the same calling convention as other functions. You can call the trap functions from your C or C++ application.

**Example**

```
#pragma vector=0x14
__trap int my_trap_function(void);
```

**See also** *Calling convention*, page 128 and *Trap functions*, page 57.

## **\_\_user**

**Syntax** Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

**Description** Functions that are declared `__user` are called from supervisor mode by the `JUSR` assembler instruction.

**Example**

```
__user int MyUserFunction(void);
```

**See also** *User functions*, page 58.



# Pragma directives

This chapter describes the pragma directives of the compiler.

The `#pragma` directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

---

## Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

| Pragma directive                     | Description                                                                                                |
|--------------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>basic_template_matching</code> | Makes a template function fully memory-attribute aware                                                     |
| <code>bitfields</code>               | Controls the order of bitfield members                                                                     |
| <code>data_alignment</code>          | Gives a variable a higher (more strict) alignment                                                          |
| <code>diag_default</code>            | Changes the severity level of diagnostic messages                                                          |
| <code>diag_error</code>              | Changes the severity level of diagnostic messages                                                          |
| <code>diag_remark</code>             | Changes the severity level of diagnostic messages                                                          |
| <code>diag_suppress</code>           | Suppresses diagnostic messages                                                                             |
| <code>diag_warning</code>            | Changes the severity level of diagnostic messages                                                          |
| <code>error</code>                   | Signals an error while parsing                                                                             |
| <code>include_alias</code>           | Specifies an alias for an include file                                                                     |
| <code>inline</code>                  | Controls inlining of a function                                                                            |
| <code>language</code>                | Controls the IAR Systems language extensions                                                               |
| <code>location</code>                | Specifies the absolute address of a variable, or places groups of functions or variables in named segments |
| <code>message</code>                 | Prints a message                                                                                           |

*Table 35: Pragma directives summary*

| Pragma directive                   | Description                                                                                     |
|------------------------------------|-------------------------------------------------------------------------------------------------|
| <code>object_attribute</code>      | Changes the definition of a variable or a function                                              |
| <code>optimize</code>              | Specifies the type and level of an optimization                                                 |
| <code>pack</code>                  | Specifies the alignment of structures and union members                                         |
| <code>__printf_args</code>         | Verifies that a function with a printf-style format string is called with the correct arguments |
| <code>required</code>              | Ensures that a symbol that is needed by another symbol is included in the linked output         |
| <code>rtmodel</code>               | Adds a runtime model attribute to the module                                                    |
| <code>__scanf_args</code>          | Verifies that a function with a scanf-style format string is called with the correct arguments  |
| <code>section</code>               | This directive is an alias for <code>#pragma segment</code>                                     |
| <code>segment</code>               | Declares a segment name to be used by intrinsic functions                                       |
| <code>STDC CX_LIMITED_RANGE</code> | Specifies whether the compiler can use normal complex mathematical formulas or not              |
| <code>STDC FENV_ACCESS</code>      | Specifies whether your source code accesses the floating-point environment or not.              |
| <code>STDC FP_CONTRACT</code>      | Specifies whether the compiler is allowed to contract floating-point expressions or not.        |
| <code>type_attribute</code>        | Changes the declaration and definitions of a variable or function                               |
| <code>vector</code>                | Specifies the vector of an interrupt or trap function                                           |

Table 35: Pragma directives summary (Continued)

**Note:** For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 319.

## Descriptions of pragma directives

This section gives detailed information about each pragma directive.

### basic\_template\_matching

|             |                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma basic_template_matching</code>                                                                                                                          |
| Description | Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That |

template function will then match the template without the modifications, see *Templates and data memory attributes*, page 162.

**Example**

```
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
    fun((int __data16 *) 0); // T = int __data16
}
```

**bitfields****Syntax**

```
#pragma bitfields={reversed|default}
```

**Parameters**

|          |                                                                                         |
|----------|-----------------------------------------------------------------------------------------|
| reversed | Bitfield members are placed from the most significant bit to the least significant bit. |
| default  | Bitfield members are placed from the least significant bit to the most significant bit. |

**Description**

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

**Example**

```
#pragma bitfields=reversed
/* Structure that uses reversed bitfields. */
struct S
{
    unsigned char error : 1;
    unsigned char size : 4;
    unsigned short code : 10;
};
#pragma bitfields=default /* Restores to default setting. */
```

**See also**

*Bitfields*, page 229.

**data\_alignment****Syntax**

```
#pragma data_alignment=expression
```

**Parameters**

|                   |                                                          |
|-------------------|----------------------------------------------------------|
| <i>expression</i> | A constant which must be a power of two (1, 2, 4, etc.). |
|-------------------|----------------------------------------------------------|

**Description** Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

**Note:** Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

## diag\_default

**Syntax** `#pragma diag_default=tag[, tag, ...]`

**Parameters** `tag` The number of a diagnostic message, for example the message number `Pe177`.

**Description** Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

**See also** *Diagnostics*, page 192.

## diag\_error

**Syntax** `#pragma diag_error=tag[, tag, ...]`

**Parameters** `tag` The number of a diagnostic message, for example the message number `Pe177`.

**Description** Use this pragma directive to change the severity level to `error` for the specified diagnostics.

**See also** *Diagnostics*, page 192.

## diag\_remark

|             |                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_remark=tag[, tag, ...]</code>                                                                     |
| Parameters  | <i>tag</i> The number of a diagnostic message, for example the message number Pe177.                                 |
| Description | Use this pragma directive to change the severity level to <code>remark</code> for the specified diagnostic messages. |
| See also    | <i>Diagnostics</i> , page 192.                                                                                       |

## diag\_suppress

|             |                                                                                      |
|-------------|--------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_suppress=tag[, tag, ...]</code>                                   |
| Parameters  | <i>tag</i> The number of a diagnostic message, for example the message number Pe117. |
| Description | Use this pragma directive to suppress the specified diagnostic messages.             |
| See also    | <i>Diagnostics</i> , page 192.                                                       |

## diag\_warning

|             |                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_warning=tag[, tag, ...]</code>                                                                     |
| Parameters  | <i>tag</i> The number of a diagnostic message, for example the message number Pe826.                                  |
| Description | Use this pragma directive to change the severity level to <code>warning</code> for the specified diagnostic messages. |
| See also    | <i>Diagnostics</i> , page 192.                                                                                        |

## error

|             |                                                                                                                                                                                                                                                                                                                                                       |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma error <i>message</i></code>                                                                                                                                                                                                                                                                                                             |
| Parameters  | <i>message</i> A string that represents the error message.                                                                                                                                                                                                                                                                                            |
| Description | Use this pragma directive to cause an error message when it is parsed. This mechanism is different from the preprocessor directive <code>#error</code> , because the <code>#pragma error</code> directive can be included in a preprocessor macro using the <code>_Pragma</code> form of the directive and only causes an error if the macro is used. |
| Example     | <pre>#if FOO_AVAILABLE #define FOO ... #else #define FOO _Pragma("error\"Foo is not available\"") #endif</pre> <p>If <code>FOO_AVAILABLE</code> is zero, an error will be signaled if the <code>FOO</code> macro is used in actual source code.</p>                                                                                                   |

## include\_alias

|             |                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma include_alias ("<i>orig_header</i>" , "<i>subst_header</i>")</code><br><code>#pragma include_alias (&lt;<i>orig_header</i>&gt; , &lt;<i>subst_header</i>&gt;)</code>                                                                                                                                                                                                            |
| Parameters  | <i>orig_header</i> The name of a header file for which you want to create an alias.<br><br><i>subst_header</i> The alias for the original header file.                                                                                                                                                                                                                                        |
| Description | Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.<br><br>This pragma directive must appear before the corresponding <code>#include</code> directives and <code>subst_header</code> must match its corresponding <code>#include</code> directive exactly. |
| Example     | <pre>#pragma include_alias (&lt;stdio.h&gt; , &lt;C:\MyHeaders\stdio.h&gt;)</pre> <pre>#include &lt;stdio.h&gt;</pre> <p>This example will substitute the relative file <code>stdio.h</code> with a counterpart located according to the specified path.</p>                                                                                                                                  |



See also *Include file search procedure*, page 190.

## inline

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

### Parameters

|                     |                                                                                          |
|---------------------|------------------------------------------------------------------------------------------|
| No parameter        | Has the same effect as the <code>inline</code> keyword.                                  |
| <code>forced</code> | Disables the compiler's heuristics and forces inlining.                                  |
| <code>never</code>  | Disables the compiler's heuristics and makes sure that the function will not be inlined. |

### Description

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

Specifying `#pragma inline=forced` will always inline the defined function. If the compiler fails to inline the function for some reason, for example due to recursion, a warning message is emitted.

Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` will enable inlining of the function also on the Medium optimization level.

See also *Inlining functions*, page 63

## language

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

### Parameters

|                       |                                                                                                                                                                          |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>extended</code> | Enables the IAR Systems language extensions from the first use of the pragma directive and onward.                                                                       |
| <code>default</code>  | From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options. |

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

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

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

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

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

Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled, but where the state before the sequence cannot be assumed to be the same as that specified by the compiler options in use:

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

**See also** `-e`, page 208 and `--strict`, page 223.

## 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 segment name; cannot be a segment 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 segment for placing either a variable or a function whose declaration follows the pragma directive. Do not place variables that would normally be in different segments (for example, variables declared as `__no_init` and variables declared as `const`) in the same named segment.

**Example**

```
#pragma location=0xFF2000
__no_init volatile char PORT1; /* PORT1 is located at address
                                0xFF2000 */

#pragma segment="FLASH"
#pragma location="FLASH"
__no_init char PORT2; /* PORT2 is located in segment FLASH */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
/* ... */
FLASH __no_init int i; /* i is placed in the FLASH segment */
```

See also *Controlling data and function placement in memory, page 173.*

## message

**Syntax** `#pragma message(message)`

**Parameters**

*message*                      The message that you want to direct to the standard output stream.

**Description**

Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

**Example**

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

## object\_attribute

**Syntax** `#pragma object_attribute=object_attribute[,object_attribute,...]`

**Parameters**

For information about object attributes that can be used with this pragma directive, see *Object attributes, page 248.*

**Description**

Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.

**Example** `#pragma object_attribute=__no_init  
char bar;`

**See also** *General syntax rules for extended keywords*, page 239.

## optimize

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

### Parameters

|                        |                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>goal</i>            | Choose between:<br><br>size, optimizes for size<br><br>balanced, optimizes balanced between speed and size<br><br>speed, optimizes for speed.<br><br>no_size_constraints, optimizes for speed, but relaxes the normal restrictions for code size expansion.                                                                                                                                 |
| <i>level</i>           | Specifies the level of optimization; choose between none, low, medium, or high.                                                                                                                                                                                                                                                                                                             |
| <i>no_optimization</i> | Disables one or several optimizations; choose between:<br><br>no_code_motion, disables code motion<br><br>no_crosscall, disables cross calls<br><br>no_crossjump, disables cross jumping<br><br>no_cse, disables common subexpression elimination<br><br>no_inline, disables function inlining<br><br>no_tbaa, disables type-based alias analysis<br><br>no_unroll, disables loop unrolling |

**Description** Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `size`, `balanced`, `speed`, and `no_size_constraints` only have effect on the `high` optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

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

**Example**

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

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

**See also**

*Fine-tuning enabled transformations*, page 179.

**pack****Syntax**

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

**Parameters**

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

**Description**

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

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

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

**See also**

*Packed structure types*, page 235

## \_\_printf\_args

|             |                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma __printf_args</code>                                                                                                                                                                                        |
| Description | Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct. |
| Example     | <pre>#pragma __printf_args int printf(char const *,...);  void PrintNumbers(unsigned short x) {     printf("%d", x); /* Compiler checks that x is an integer */ }</pre>                                                   |

## required

|             |                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma required=<i>symbol</i></code>                                                                                                                                                                                                                                                                                                                                                       |
| Parameters  | <i>symbol</i> Any statically linked function or variable.                                                                                                                                                                                                                                                                                                                                         |
| Description | <p>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.</p> <p>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 segment it resides in.</p> |
| Example     | <pre>const char copyright[] = "Copyright by me";  #pragma required=copyright int main() {     /* Do something here. */ }</pre> <p>Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.</p>                                                                                                                           |

## rtmodel

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma rtmodel="key", "value"</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Parameters  | <p><code>"key"</code> A text string that specifies the runtime model attribute.</p> <p><code>"value"</code> A text string that specifies the value of the runtime model attribute. Using the special value <code>*</code> is equivalent to not defining the attribute at all.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description | <p>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.</p> <p>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 <code>*</code>. It can, however, be useful to state explicitly that the module can handle any runtime model.</p> <p>A module can have several runtime model definitions.</p> <p><b>Note:</b> 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.</p> |
| Example     | <pre>#pragma rtmodel="I2C", "ENABLED"</pre> <p>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.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| See also    | <i>Checking module consistency</i> , page 110.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

## \_\_scanf\_args

|             |                                                                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma __scanf_args</code>                                                                                                                                                                                                      |
| Description | Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example <code>%d</code> ) 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;
}
```

## segment

## Syntax

```
#pragma segment="NAME" [__memoryattribute] [align]
alias
#pragma section="NAME" [__memoryattribute] [align]
```

## Parameters

|                          |                                                                                                                              |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <i>NAME</i>              | The name of the segment.                                                                                                     |
| <i>__memoryattribute</i> | An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used. |
| <i>align</i>             | Specifies an alignment for the segment. The value must be a constant integer expression to the power of two.                 |

## Description

Use this pragma directive to define a segment name that can be used by the segment operators `__segment_begin`, `__segment_end`, and `__segment_size`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

The *align* and the *\_\_memoryattribute* parameters are only relevant when used together with the segment operators `__segment_begin`, `__segment_end`, and `__segment_size`. If you consider using *align* on an individual variable to achieve a higher alignment, you must instead use the `#pragma data_alignment` directive.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

```
void __memoryattribute *.
```

## Example

```
#pragma segment="MYDATA20" __data20 2
```



See also *Dedicated segment operators*, page 149. For more information about segments, see the chapter *Placing code and data*.

## STDC CX\_LIMITED\_RANGE

Syntax `#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}`

|            |         |                                                    |
|------------|---------|----------------------------------------------------|
| Parameters | ON      | Normal complex mathematic formulas can be used.    |
|            | OFF     | Normal complex mathematic formulas cannot be used. |
|            | DEFAULT | Sets the default behavior, that is OFF.            |

Description Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for \* (multiplication), / (division), and abs.

**Note:** This directive is required by Standard C. The directive is recognized but has no effect in the compiler.

## STDC FENV\_ACCESS

Syntax `#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}`

|            |         |                                                                                                                |
|------------|---------|----------------------------------------------------------------------------------------------------------------|
| Parameters | ON      | Source code accesses the floating-point environment. Note that this argument is not supported by the compiler. |
|            | OFF     | Source code does not access the floating-point environment.                                                    |
|            | DEFAULT | Sets the default behavior, that is OFF.                                                                        |

Description Use this pragma directive to specify whether your source code accesses the floating-point environment or not.

**Note:** This directive is required by Standard C.

## STDC FP\_CONTRACT

Syntax `#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}`

|             |                                                                                                                                                               |                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Parameters  | ON                                                                                                                                                            | The compiler is allowed to contract floating-point expressions.                                                               |
|             | OFF                                                                                                                                                           | The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler. |
|             | DEFAULT                                                                                                                                                       | Sets the default behavior, that is ON.                                                                                        |
| Description | Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C. |                                                                                                                               |
| Example     | <code>#pragma STDC FP_CONTRACT=ON</code>                                                                                                                      |                                                                                                                               |

## type\_attribute

|             |                                                                                                                                                                                                                                                                                                                                                                      |  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>#pragma type_attribute=type_attribute[, type_attribute, ...]</code>                                                                                                                                                                                                                                                                                            |  |
| Parameters  | For information about type attributes that can be used with this pragma directive, see <i>Type attributes</i> , page 239.                                                                                                                                                                                                                                            |  |
| Description | Use this pragma directive to specify IAR-specific <i>type attributes</i> , which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects.<br><br>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, an <code>int</code> object with the memory attribute <code>__data20</code> is defined:<br><br><code>#pragma type_attribute=__data20<br/>int x;</code><br><br>This declaration, which uses extended keywords, is equivalent:<br><br><code>__data20 int x;</code>                                                                                     |  |
| See also    | The chapter <i>Extended keywords</i> for more information.                                                                                                                                                                                                                                                                                                           |  |

## vector

|        |                                                              |  |
|--------|--------------------------------------------------------------|--|
| Syntax | <code>#pragma vector=vector1[, vector2, vector3, ...]</code> |  |
|--------|--------------------------------------------------------------|--|

|             |                                                                                                                                                                                                 |                                                        |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| Parameters  | <i>vectorN</i>                                                                                                                                                                                  | The vector number(s) of an interrupt or trap function. |
| Description | Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function. |                                                        |
| Example     | <pre>#pragma vector=0x14 __interrupt void my_handler(void);</pre>                                                                                                                               |                                                        |



# 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:

| Intrinsic function                          | Description                                        |
|---------------------------------------------|----------------------------------------------------|
| <code>__adjust_return_address</code>        | Adjusts the return address for exceptions          |
| <code>__clear_processor_register_bit</code> | Clears a bit in a processor register               |
| <code>__clear_PSR_I_bit</code>              | Clears the interrupt bit in the processor register |
| <code>__disable_interrupt</code>            | Disables interrupts                                |
| <code>__enable_interrupt</code>             | Enables interrupts                                 |
| <code>__enable_interrupt_wait</code>        | Inserts an EIWAIT instruction                      |
| <code>__get_interrupt_state</code>          | Returns the interrupt state                        |
| <code>__get_processor_register</code>       | Reads the specified processor register             |
| <code>__mac_q15</code>                      | Multiplies and accumulates the Q15 format          |
| <code>__mac_signed</code>                   | Multiplies and accumulates signed integer          |
| <code>__mac_unsigned</code>                 | Multiplies and accumulates unsigned integer        |
| <code>__memcpy_generic</code>               | Copies a number of bytes between locations         |
| <code>__memset_generic</code>               | Sets all bytes in a memory block                   |
| <code>__no_operation</code>                 | Inserts a NOP instruction                          |

Table 36: Intrinsic functions summary

| Intrinsic function                        | Description                                                   |
|-------------------------------------------|---------------------------------------------------------------|
| <code>__raise_exception</code>            | Inserts an EXCP instruction                                   |
| <code>__require</code>                    | Sets a constant literal, available for backward compatibility |
| <code>__set_interrupt_state</code>        | Restores the interrupt state                                  |
| <code>__set_processor_register</code>     | Sets the value of a processor register                        |
| <code>__set_processor_register_bit</code> | Sets a bit in a processor register                            |
| <code>__set_PSR_I_bit</code>              | Sets the interrupt bit in the processor register              |
| <code>__wait_for_interrupt</code>         | Inserts a WAIT instruction                                    |

Table 36: Intrinsic functions summary (Continued)

## Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

### `__adjust_return_address`

Syntax `void __adjust_return_address(long);`

Description When a trap or interrupt occurs in the CR16C microprocessor, the return address is stored on the interrupt stack. The intrinsic function `__adjust_return_address` can be used for adjusting this value.

Note that trap functions declared without the `__noadjust` function object type attribute will adjust their return address by two bytes.

### `__clear_processor_register_bit`

Syntax `__clear_processor_register_bit(int, unsigned long);`

Description Clears a bit in a processor register. The bit should be a bit mask with the corresponding bit set. The header file `intrinsics.h` defines the registers `__PSR` and `__CFG`. It also defines the following processor bits:

|                      |                      |                        |
|----------------------|----------------------|------------------------|
| <code>__PSR_C</code> | <code>__PSR_Z</code> | <code>__CFG_DC</code>  |
| <code>__PSR_T</code> | <code>__PSR_N</code> | <code>__CFG_LDC</code> |
| <code>__PSR_L</code> | <code>__PSR_E</code> | <code>__CFG_IC</code>  |

```

__PSR_U          __PSR_P          __CFG_LIC
__PSR_F          __PSR_I          __CFG_EC
                                   __CFG_SR

```

Example `__clear_processor_register_bit(__CFG, __CFG_DC);`

## **\_\_clear\_PSR\_I\_bit**

Syntax `void __clear_PSR_I_bit(void);`

Description Disables interrupts by clearing the interrupt bit in the processor status register.

## **\_\_disable\_interrupt**

Syntax `void __disable_interrupt(void);`

Description Disables interrupts by inserting the `DI` instruction.

## **\_\_enable\_interrupt**

Syntax `void __enable_interrupt(void);`

Description Enables interrupts by inserting the `EI` instruction.

## **\_\_enable\_interrupt\_wait**

Syntax `void __enable_interrupt_wait(void);`

Description Enables interrupts and waits for an interrupt by inserting an `EIWAIT` instruction.

## **\_\_get\_interrupt\_state**

Syntax `__istate_t __get_interrupt_state(void);`

Description Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

**Example**

```
#include "intrinsics.h"

void CriticalFn()
{
    __istate_t s = __get_interrupt_state();
    __disable_interrupt();

    /* Do something here. */

    __set_interrupt_state(s);
}
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

## **\_\_get\_processor\_register**

**Syntax**

```
unsigned long __get_processor_register(int);
```

**Description**

Reads the specified processor register. The available processor registers are:

```
__R0 - __R15
__CAR0
__CAR1
__CFG
__DBS
__DCR
__DSR
__INTBASE
__ISP
__PSR
__USP
```

The registers are defined in the `intrinsics.h` header file.

## **\_\_mac\_q15**

**Syntax**

```
long __mac_q15(short a, short b, long c);
```

**Description**

Multiplies and accumulates fractional numbers in the Q15 format using the `MACQW` instruction. Returns the sum of `c` and the product of `a` and `b`.



**\_\_mac\_signed**

|             |                                                                                      |
|-------------|--------------------------------------------------------------------------------------|
| Syntax      | <code>long __mac_signed(short, short, long);</code>                                  |
| Description | Multiplies and accumulates signed integers using the <code>MACSW</code> instruction. |

**\_\_mac\_unsigned**

|             |                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __mac_unsigned(unsigned short, unsigned short,<br/>unsigned long);</code> |
| Description | Multiplies and accumulates unsigned integers using the <code>MACUW</code> instruction.        |

**\_\_memcpy\_generic**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __data32 * __memcpy_generic(void __data32 *,<br/>void __data32 const *, long);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Description | <p>Copies a number of bytes from one location to another. The first parameter should point to the location to which the data should be copied, the second parameter should point to the data to be copied. The last argument describes the number of bytes that should be copied. The function returns a pointer to the destination location.</p> <p>This function behaves identically to the standard function <code>memcpy</code> with the exception that the pointers can point to any location in memory. Remember that the functions in the standard library use the default pointer type, which might not be able to reach the entire memory depending on the data model used.</p> |

**\_\_memset\_generic**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __data32 * __memset_generic(void __data32 *, int, long);</code>                                                                                                                                                                                                                                                                                                                                                               |
| Description | <p>Sets all bytes in a memory block to the value of the second parameter.</p> <p>The memory block should be pointed to by the first parameter. The third parameter specifies the size of the block. This function returns a pointer to the beginning of the block.</p> <p>This function behaves identically to the standard function <code>memset</code>, with the exception that the memory block can be placed anywhere in memory.</p> |

## **\_\_no\_operation**

Syntax `void __no_operation(void);`

Description Inserts a NOP instruction.

## **\_\_raise\_exception**

Syntax `void __raise_exception(int);`

Description Inserts an EXCP instruction. `int` is the exception vector number.

## **\_\_set\_interrupt\_state**

Syntax `void __set_interrupt_state(__istate_t);`

Description Restores the interrupt state to a value previously returned by the `__get_interrupt_state` function.

For information about the `__istate_t` type, see `__get_interrupt_state`, page 271.

## **\_\_set\_processor\_register**

Syntax `void __set_processor_register(int, unsigned long);`

Description Sets a processor register to a specific value. The registers supported are the same as for `__get_processor_register`, page 272.

## **\_\_set\_processor\_register\_bit**

Syntax `void __set_processor_register_bit(int, unsigned long);`

Description Sets a bit in a processor register. The bit should be a bit mask with the corresponding bit set.

Example `__set_processor_register_bit(__CFG, __CFG_DC);`

See also `__clear_processor_register_bit`, page 270

## **\_\_set\_PSR\_I\_bit**

|             |                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __set_PSR_I_bit(int, unsigned long);</code>                                           |
| Description | Sets a bit in a processor register. The bit should be a bit mask with the corresponding bit set. |
| Example     | <code>__set_processor_register_bit(__CFG, __CFG_DC);</code>                                      |
| See also    | <i>__clear_processor_register_bit</i> , page 270                                                 |

## **\_\_wait\_for\_interrupt**

|             |                                               |
|-------------|-----------------------------------------------|
| Syntax      | <code>void __wait_for_interrupt(void);</code> |
| Description | Inserts a <code>WAIT</code> instruction.      |



# 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 CR16C adheres to Standard C. The compiler also makes these preprocessor-related features available to you:

- **Predefined preprocessor symbols**  
These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For more information, see *Description of predefined preprocessor symbols*, page 278.
- **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 202.
- **Preprocessor extensions**  
There are several preprocessor extensions, for example many `pragma` directives; for more information, see the chapter *Pragma directives*. For information about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 282.
- **Preprocessor output**  
Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 220.

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

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

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile", "rt");
```

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.

---

## Description of predefined preprocessor symbols

This section lists and describes the preprocessor symbols.

### **\_\_BASE\_FILE\_\_**

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

**See also** See also `__FILE__`, page 279, and `--no_path_in_file_macros`, page 214.

### **\_\_BUILD\_NUMBER\_\_**

**Description** A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.

### **\_\_CODE\_MODEL\_\_**

**Description** An integer that identifies the code model in use. The symbol reflects the `--code_model` option and is defined to `__CODE_MODEL_NORMAL__` or `__CODE_MODEL_SHORT__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.

### **\_\_cplusplus\_\_**

**Description** An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

### **\_\_DATA\_MODEL\_\_**

**Description** An integer that identifies the data model in use. The symbol reflects the `--data_model` option and is defined to `__DATA_MODEL_SMALL__`, `__DATA_MODEL_MEDIUM__`, `__DATA_MODEL_LARGE__`, `__DATA_MODEL_HUGE__`, or `__DATA_MODEL_INDEXED__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.

**\_\_DATE\_\_**

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

This symbol is required by Standard C.

**\_\_DOUBLE\_\_**

Description An integer that identifies the size of the data type `double`. The symbol is defined to 64

**\_\_embedded\_cplusplus**

Description An integer which is defined to 1 when the compiler runs in 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.

This symbol is required by Standard C.

**\_\_FILE\_\_**

Description A string that identifies the name of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

See also `__BASE_FILE__`, page 278, and `--no_path_in_file_macros`, page 214.

**\_\_func\_\_**

Description A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

This symbol is required by Standard C.

See also `-e`, page 208. See also `__PRETTY_FUNCTION__`, page 280.

**\_\_FUNCTION\_\_**

Description                    A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

See also                         -e, page 208. See also `__PRETTY_FUNCTION__`, page 280.

**\_\_IAR\_SYSTEMS\_ICC\_\_**

Description                    An integer that identifies the IAR compiler platform. The current value is 8. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.

**\_\_ICC\_CR16C\_\_**

Description                    An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for CR16C.

**\_\_INDEXED\_ENABLED\_\_**

Description                    A symbol that is defined to 1 when it is possible to use the memory attributes `__ix4` and `__ix20`.

**\_\_LINE\_\_**

Description                    An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

**\_\_LITTLE\_ENDIAN\_\_**

Description                    An integer that reflects the byte order and is defined to 1 (little-endian).

**\_\_PRETTY\_FUNCTION\_\_**

Description                    A predefined string identifier that is initialized with the function name, including parameter types and return type, of the function in which the symbol is used, for



example `"void func(char)".` This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

See also

`-e`, page 208. See also `__func__`, page 279.

## **\_\_STDC\_\_**

Description

An integer that is set to 1, which means the compiler adheres to Standard C. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to Standard C.\*

This symbol is required by Standard C.

## **\_\_STDC\_VERSION\_\_**

Description

An integer that identifies the version of the C standard in use. The symbol expands to 199901L, unless the `--c89` compiler option is used in which case the symbol expands to 199409L. This symbol does not apply in EC++ mode.

This symbol is required by Standard C.

## **\_\_SUBVERSION\_\_**

Description

An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.

## **\_\_TIME\_\_**

Description

A string that identifies the time of compilation in the form "hh:mm:ss".

This symbol is required by Standard C.

## **\_\_VER\_\_**

Description

An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: (100 \* the major version number + the minor version number). For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

---

## Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and Standard C directives.

### NDEBUG

#### Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

#### See also

*Assert*, page 109.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

### #warning message

#### Syntax

`#warning message`

where *message* can be any string.

#### Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.

# Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

---

## Library overview

The compiler comes with two different libraries:

- The IAR DLIB Library is a complete library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format and it does not support C++.

For more information about customization, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For more information about library functions, see the chapter *Implementation-defined behavior for Standard C* in this guide.

## HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

## LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic project configuration*, page 37. The linker will include only those routines that are required—directly or indirectly—by your application.

## ALTERNATIVE MORE ACCURATE LIBRARY FUNCTIONS

The default implementation of `cos`, `sin`, `tan`, and `pow` is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named `__iar_xxx_accuratef` for float variants of the functions and `__iar_xxx_accuratel` for long double variants of the functions, and where `xxx` is `cos`, `sin`, etc.

To use any of these more accurate versions, use the `-e` linker option.

## REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `printf`, `sprintf`, `scanf`, `sscanf`, `getchar`, and `putchar`.

For the CLIB library, the `qsort` function and functions that use files in some way are non-reentrant. This includes `printf`, `scanf`, `getchar`, and `putchar`. However, the functions `sprintf` and `sscanf` are reentrant.

Functions that can set `errno` are not reentrant, because an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. This applies to math and string conversion functions, among others.

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

## THE LONGJMP FUNCTION

A `longjmp` is in effect a jump to a previously defined `setjmp`. Any variable length arrays or C++ objects residing on the stack during stack unwinding will not be destroyed. This can lead to resource leaks or incorrect application behavior.

---

## IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior for Standard C* in this guide.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code, see the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of CR16C features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, see *Added C functionality*, page 289.

## C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Using C*.

This table lists the C header files:

| Header file            | Usage                                           |
|------------------------|-------------------------------------------------|
| <code>assert.h</code>  | Enforcing assertions when functions execute     |
| <code>complex.h</code> | Computing common complex mathematical functions |

Table 37: Traditional Standard C header files—DLIB

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

Table 37: Traditional Standard C header files—DLIB (Continued)

## C++ HEADER FILES

This section lists the C++ header files:

- The C++ library header files  
The header files that constitute the Embedded C++ library.
- The C++ standard template library (STL) header files  
The header files that constitute STL for the Extended Embedded C++ library.
- The C++ C header files  
The C++ header files that provide the resources from the C library.

## The C++ library header files

This table lists the header files that can be used in Embedded C++:

| Header file               | Usage                                                                             |
|---------------------------|-----------------------------------------------------------------------------------|
| <code>complex</code>      | Defining a class that supports complex arithmetic                                 |
| <code>fstream</code>      | Defining several I/O stream classes that manipulate external files                |
| <code>iomanip</code>      | Declaring several I/O stream manipulators that take an argument                   |
| <code>ios</code>          | Defining the class that serves as the base for many I/O streams classes           |
| <code>iosfwd</code>       | Declaring several I/O stream classes before they are necessarily defined          |
| <code>iostream</code>     | Declaring the I/O stream objects that manipulate the standard streams             |
| <code>istream</code>      | Defining the class that performs extractions                                      |
| <code>new</code>          | Declaring several functions that allocate and free storage                        |
| <code>ostream</code>      | Defining the class that performs insertions                                       |
| <code>sstream</code>      | Defining several I/O stream classes that manipulate string containers             |
| <code>streambuf</code>    | Defining classes that buffer I/O stream operations                                |
| <code>string</code>       | Defining a class that implements a string container                               |
| <code>stringstream</code> | Defining several I/O stream classes that manipulate in-memory character sequences |

Table 38: C++ header files

## The C++ standard template library (STL) header files

The following table lists the standard template library (STL) header files that can be used in Extended Embedded C++:

| Header file             | Description                                            |
|-------------------------|--------------------------------------------------------|
| <code>algorithm</code>  | Defines several common operations on sequences         |
| <code>deque</code>      | A deque sequence container                             |
| <code>functional</code> | Defines several function objects                       |
| <code>hash_map</code>   | A map associative container, based on a hash algorithm |
| <code>hash_set</code>   | A set associative container, based on a hash algorithm |
| <code>iterator</code>   | Defines common iterators, and operations on iterators  |
| <code>list</code>       | A doubly-linked list sequence container                |
| <code>map</code>        | A map associative container                            |
| <code>memory</code>     | Defines facilities for managing memory                 |
| <code>numeric</code>    | Performs generalized numeric operations on sequences   |

Table 39: Standard template library header files

| Header file          | Description                             |
|----------------------|-----------------------------------------|
| <code>queue</code>   | A queue sequence container              |
| <code>set</code>     | A set associative container             |
| <code>slist</code>   | A singly-linked list sequence container |
| <code>stack</code>   | A stack sequence container              |
| <code>utility</code> | Defines several utility components      |
| <code>vector</code>  | A vector sequence container             |

Table 39: Standard template library header files (Continued)

### Using Standard C libraries in C++

The C++ library works in conjunction with some of the header files from the Standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

This table shows the new header files:

| Header file            | Usage                                                              |
|------------------------|--------------------------------------------------------------------|
| <code>cassert</code>   | Enforcing assertions when functions execute                        |
| <code>cctype</code>    | Classifying characters                                             |
| <code>cerrno</code>    | Testing error codes reported by library functions                  |
| <code>cfloat</code>    | Testing floating-point type properties                             |
| <code>cinttypes</code> | Defining formatters for all types defined in <code>stdint.h</code> |
| <code>climits</code>   | Testing integer type properties                                    |
| <code>locale</code>    | Adapting to different cultural conventions                         |
| <code>cmath</code>     | Computing common mathematical functions                            |
| <code>csetjmp</code>   | Executing non-local goto statements                                |
| <code>csignal</code>   | Controlling various exceptional conditions                         |
| <code>cstdarg</code>   | Accessing a varying number of arguments                            |
| <code>cstdbool</code>  | Adds support for the <code>bool</code> data type in C.             |
| <code>cstddef</code>   | Defining several useful types and macros                           |
| <code>stdint</code>    | Providing integer characteristics                                  |
| <code>stdio</code>     | Performing input and output                                        |
| <code>stdlib</code>    | Performing a variety of operations                                 |
| <code>cstring</code>   | Manipulating several kinds of strings                              |
| <code>ctime</code>     | Converting between various time and date formats                   |

Table 40: New Standard C header files—DLIB



| Header file         | Usage                       |
|---------------------|-----------------------------|
| <code>wchar</code>  | Support for wide characters |
| <code>wctype</code> | Classifying wide characters |

Table 40: New Standard C header files—DLIB (Continued)

## LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

## ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality.

The following include files provide these features:

- `fcntl.h`
- `stdio.h`
- `stdlib.h`
- `string.h`

### **fcntl.h**

In `fcntl.h`, trap handling support for floating-point numbers is defined with the functions `fegetrapenable` and `fegettrapdisable`.

### **stdio.h**

These functions provide additional I/O functionality:

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

**string.h**

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

|                          |                                                |
|--------------------------|------------------------------------------------|
| <code>strdup</code>      | Duplicates a string on the heap.               |
| <code>strcasecmp</code>  | Compares strings case-insensitive.             |
| <code>strncasecmp</code> | Compares strings case-insensitive and bounded. |
| <code>strlen</code>      | Bounded string length.                         |

**SYMBOLS USED INTERNALLY BY THE LIBRARY**

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__code`

This symbol is used as a memory attribute internally by the compiler, and might have to be used as an argument in certain templates.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor`, `__has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

**Note:** The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

## IAR CLIB Library

The IAR CLIB Library provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- Standard C library definitions available for user programs. These are documented in this chapter.
- The system startup code; see the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of CR16C features. See the chapter *Intrinsic functions* for more information.

### LIBRARY DEFINITIONS SUMMARY

This table lists the header files specific to the CLIB library:

| Header file            | Description                                                                                                               |
|------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>assert.h</code>  | Assertions                                                                                                                |
| <code>ctype.h*</code>  | Character handling                                                                                                        |
| <code>errno.h</code>   | Error return values                                                                                                       |
| <code>float.h</code>   | Limits and sizes of floating-point types                                                                                  |
| <code>iccbutl.h</code> | Low-level routines                                                                                                        |
| <code>limits.h</code>  | Limits and sizes of integral types                                                                                        |
| <code>math.h</code>    | Mathematics                                                                                                               |
| <code>setjmp.h</code>  | Non-local jumps                                                                                                           |
| <code>stdarg.h</code>  | Variable arguments                                                                                                        |
| <code>stdbool.h</code> | Adds support for the <code>bool</code> data type in C                                                                     |
| <code>stddef.h</code>  | Common definitions including <code>size_t</code> , <code>NULL</code> , <code>ptrdiff_t</code> , and <code>offsetof</code> |
| <code>stdio.h</code>   | Input/output                                                                                                              |
| <code>stdlib.h</code>  | General utilities                                                                                                         |
| <code>string.h</code>  | String handling                                                                                                           |

Table 41: IAR CLIB Library header files

\* The functions `isxxx`, `toupper`, and `tolower` declared in the header file `ctype.h` evaluate their argument more than once. This is not according to the ISO/ANSI standard.



# Segment reference

The compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For more information about segments, see the chapter *Placing code and data*.

---

## Summary of segments

The table below lists the segments that are available in the compiler:

| Segment     | Description                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------|
| ABSOLUTE_C  | Holds located constant data.                                                                          |
| ABSOLUTE_N  | Holds located uninitialized data.                                                                     |
| CHECKSUM    | Holds the checksum generated by the linker.                                                           |
| CODE        | Holds the program code.                                                                               |
| CSTACK      | Holds the stack used by C or C++ programs.                                                            |
| CSTART      | Holds the startup code.                                                                               |
| DATA16_C    | Holds <code>__data16</code> constant data.                                                            |
| DATA16_HEAP | Holds the heap used for dynamically allocated data in data16 memory.                                  |
| DATA16_I    | Holds <code>__data16</code> static and global initialized variables.                                  |
| DATA16_ID   | Holds initial values for <code>__data16</code> static and global variables in <code>DATA16_I</code> . |
| DATA16_N    | Holds <code>__no_init __data16</code> static and global variables.                                    |
| DATA16_Z    | Holds zero-initialized <code>__data16</code> static and global variables.                             |
| DATA20_C    | Holds <code>__data20</code> constant data.                                                            |
| DATA20_I    | Holds <code>__data20</code> static and global initialized variables.                                  |
| DATA20_ID   | Holds initial values for <code>__data20</code> static and global variables in <code>DATA20_I</code> . |
| DATA20_N    | Holds <code>__no_init __data20</code> static and global variables.                                    |
| DATA20_Z    | Holds zero-initialized <code>__data20</code> static and global variables.                             |
| DATA24_C    | Holds <code>__data24</code> constant data.                                                            |

Table 42: Segment summary

| Segment     | Description                                                                                                                                                |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DATA24_I    | Holds <code>__data24</code> static and global initialized variables.                                                                                       |
| DATA24_ID   | Holds initial values for <code>__data24</code> static and global variables in <code>DATA24_I</code> .                                                      |
| DATA24_N    | Holds <code>__no_init __data24</code> static and global variables.                                                                                         |
| DATA24_Z    | Holds zero-initialized <code>__data24</code> static and global variables.                                                                                  |
| DATA32_C    | Holds <code>__data32</code> constant data.                                                                                                                 |
| DATA32_HEAP | Holds the heap used for dynamically allocated data in <code>data32</code> memory.                                                                          |
| DATA32_I    | Holds <code>__data32</code> static and global initialized variables.                                                                                       |
| DATA32_ID   | Holds initial values for <code>__data32</code> static and global variables in <code>DATA32_I</code> .                                                      |
| DATA32_N    | Holds <code>__no_init __data32</code> static and global variables.                                                                                         |
| DATA32_Z    | Holds zero-initialized <code>__data32</code> static and global variables.                                                                                  |
| DIFUNCT     | Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before <code>main</code> is called.                 |
| IDINIT      | Holds information about the segments containing non-zero initialized data.                                                                                 |
| INDEX_BASE  | Holds an empty segment that is needed in front of the segments of the <code>INDEX4</code> , <code>INDEX5</code> , and <code>INDEX20</code> segment groups. |
| INDEX4_I    | Holds initialized <code>__ix4</code> data.                                                                                                                 |
| INDEX4_ID   | Holds <code>__ix4</code> data that is copied to <code>INDEX4_I</code> by <code>cstartup</code> .                                                           |
| INDEX4_N    | Holds uninitialized <code>__ix4</code> data.                                                                                                               |
| INDEX4_Z    | Holds zero-initialized <code>__ix4</code> data.                                                                                                            |
| INDEX5_I    | Holds initialized <code>__ix4</code> data.                                                                                                                 |
| INDEX5_ID   | Holds <code>__ix4</code> data that is copied to <code>INDEX5_I</code> by <code>cstartup</code> .                                                           |
| INDEX5_N    | Holds uninitialized <code>__ix4</code> data.                                                                                                               |
| INDEX5_Z    | Holds zero-initialized <code>__ix4</code> data.                                                                                                            |
| INDEX20_C   | Holds <code>__ix20</code> constant data, including string literals.                                                                                        |
| INDEX20_I   | Holds initialized <code>__ix20</code> data.                                                                                                                |
| INDEX20_ID  | Holds <code>__ix20</code> data that is copied to <code>INDEX20_I</code> by <code>cstartup</code> .                                                         |
| INDEX20_N   | Holds uninitialized <code>__ix20</code> data.                                                                                                              |
| INDEX20_Z   | Holds zero-initialized <code>__ix20</code> data.                                                                                                           |
| INTVEC      | Contains the reset and interrupt vectors.                                                                                                                  |
| ISTACK      | Holds the stack used by interrupts and exceptions.                                                                                                         |

Table 42: Segment summary (Continued)

| Segment | Description                                                    |
|---------|----------------------------------------------------------------|
| ZINIT   | Holds initializing information used by <code>cstartup</code> . |

Table 42: Segment summary (Continued)

## Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives `-z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—`CODE`, `CONST`, or `DATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 5, *XLINK segment memory types*, page 66.

For information about the `-z` and the `-P` directives, see the *IAR Linker and Library Tools Reference Guide*.

For information about how to define segments in the linker configuration file, see *Customizing the linker configuration file*, page 66.

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

### ABSOLUTE\_C

Description

Holds located constant data.

*Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker configuration file.

### ABSOLUTE\_N

Description

Holds `__no_init` located data.

*Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker configuration file.

## CHECKSUM

|                     |                                                                                                                                                                                                     |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the checksum bytes generated by the linker. This segment also holds the <code>__checksum</code> symbol. Note that the size of this segment is affected by the linker option <code>-J</code> . |
| Segment memory type | CONST                                                                                                                                                                                               |
| Memory placement    | This segment can be placed anywhere in ROM memory.                                                                                                                                                  |
| Access type         | Read-only                                                                                                                                                                                           |

## CODE

|                     |                                                                |
|---------------------|----------------------------------------------------------------|
| Description         | Holds program code, except the code for system initialization. |
| Segment memory type | CODE                                                           |
| Memory placement    | This segment can be placed anywhere in memory.                 |
| Access type         | Read-only                                                      |

## CSTACK

|                     |                                                                                                                                                             |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the internal data stack.                                                                                                                              |
| Segment memory type | DATA                                                                                                                                                        |
| Memory placement    | In the Small data model, this segment must be placed in the first 64 Kbytes of memory. In other data models, this segment can be placed anywhere in memory. |
| Access type         | Read-write                                                                                                                                                  |
| See also            | <i>The stack</i> , page 72.                                                                                                                                 |

## CSTART

|             |                         |
|-------------|-------------------------|
| Description | Holds the startup code. |
|-------------|-------------------------|



This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

|                     |                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------|
| Segment memory type | CODE                                                                                    |
| Memory placement    | This segment must be placed at the address where the chip starts executing after reset. |
| Access type         | Read-only                                                                               |

## DATA16\_C

|                     |                                                               |
|---------------------|---------------------------------------------------------------|
| Description         | Holds <code>__data16</code> constant data.                    |
| Segment memory type | CONST                                                         |
| Memory placement    | This segment must be placed in the first 64 Kbytes of memory. |
| Access type         | Read-only                                                     |

## DATA16\_HEAP

|                     |                                                                                                                                                                                                                                     |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the heap used for dynamically allocated data in <code>data16</code> memory, in other words data allocated by <code>data16_malloc</code> and <code>data16_free</code> , and in C++, <code>new</code> and <code>delete</code> . |
| Segment memory type | DATA                                                                                                                                                                                                                                |
| Memory placement    | This segment must be placed in the first 64 Kbytes of memory.                                                                                                                                                                       |
| Access type         | Read-write                                                                                                                                                                                                                          |
| See also            | <i>The heap</i> , page 73 and <i>New and Delete operators</i> , page 159.                                                                                                                                                           |

## DATA16\_I

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__data16</code> static and global initialized variables initialized by copying from the segment <code>DATA16_ID</code> at application startup.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                     |                                                               |
|---------------------|---------------------------------------------------------------|
| Segment memory type | DATA                                                          |
| Memory placement    | This segment must be placed in the first 64 Kbytes of memory. |
| Access type         | Read-write                                                    |

## DATA16\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__data16</code> static and global variables in the <code>DATA16_I</code> segment. These values are copied from <code>DATA16_ID</code> to <code>DATA16_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

## DATA16\_N

|                     |                                                                    |
|---------------------|--------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __data16</code> variables. |
| Segment memory type | DATA                                                               |
| Memory placement    | This segment must be placed in the first 64 Kbytes of memory.      |
| Access type         | Read-write                                                         |

## DATA16\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__data16</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                           |

Memory placement This segment must be placed in the first 64 Kbytes of memory.

Access type Read-write

## DATA20\_C

Description Holds `__data20` constant data.

Segment memory type `CONST`

Memory placement This segment must be placed in the range `0x0-0xEFFFF`.

Access type Read-only

## DATA20\_I

Description Holds `__data20` static and global initialized variables initialized by copying from the segment `DATA20_ID` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type `DATA`

Memory placement This segment must be placed in the range `0x0-0xEFFFF`.

Access type Read-write

## DATA20\_ID

Description Holds initial values for `__data20` static and global variables in the `DATA20_I` segment. These values are copied from `DATA20_ID` to `DATA20_I` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type `CONST`

Memory placement This segment can be placed anywhere in memory.

Access type Read-only

## DATA20\_N

Description Holds static and global `__no_init __data20` variables.

Segment memory type DATA

Memory placement This segment must be placed in the range `0x0-0xEFFFFF`.

Access type Read-write

## DATA20\_Z

Description Holds zero-initialized `__data20` static and global variables. The contents of this segment is declared by the system startup code.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type DATA

Memory placement This segment must be placed in the range `0x0-0xEFFFFF`.

Access type Read-write

## DATA24\_C

Description Holds `__data24` constant data.

Segment memory type CONST

Memory placement This segment must be placed in the range `0x0-0xFFFFFFFF`.

Access type Read-only

## DATA24\_I

Description Holds `__data24` static and global initialized variables initialized by copying from the segment `DATA24_ID` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type

DATA

Memory placement

This segment must be placed in the range `0x0-0xFFFFFFFF`.

Access type

Read-write

## DATA24\_ID

Description

Holds initial values for `__data24` static and global variables in the `DATA24_I` segment. These values are copied from `DATA24_ID` to `DATA24_I` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type

CONST

Memory placement

This segment can be placed anywhere in memory.

Access type

Read-only

## DATA24\_N

Description

Holds static and global `__no_init __data24` variables.

Segment memory type

DATA

Memory placement

This segment must be placed in the range `0x0-0xFFFFFFFF`.

Access type

Read-write

## DATA24\_Z

Description

Holds zero-initialized `__data24` static and global variables. The contents of this segment is declared by the system startup code.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

|                     |                                                                        |
|---------------------|------------------------------------------------------------------------|
| Segment memory type | DATA                                                                   |
| Memory placement    | This segment must be placed in the range <code>0x0-0xFFFFFFFF</code> . |
| Access type         | Read-write                                                             |

## DATA32\_C

|                     |                                                                       |
|---------------------|-----------------------------------------------------------------------|
| Description         | Holds <code>__data32</code> constant data, including string literals. |
| Segment memory type | CONST                                                                 |
| Memory placement    | This segment can be placed anywhere in memory.                        |
| Access type         | Read-only                                                             |

## DATA32\_HEAP

|                     |                                                                                                                                                                                                                                     |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the heap used for dynamically allocated data in <code>data32</code> memory, in other words data allocated by <code>data32_malloc</code> and <code>data32_free</code> , and in C++, <code>new</code> and <code>delete</code> . |
| Segment memory type | DATA                                                                                                                                                                                                                                |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                      |
| Access type         | Read-write                                                                                                                                                                                                                          |
| See also            | <i>The heap</i> , page 73 and <i>New and Delete operators</i> , page 159.                                                                                                                                                           |

## DATA32\_I

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__data32</code> static and global initialized variables initialized by copying from the segment <code>DATA32_ID</code> at application startup.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                     |                                                |
|---------------------|------------------------------------------------|
| Segment memory type | DATA                                           |
| Memory placement    | This segment can be placed anywhere in memory. |
| Access type         | Read-write                                     |

## DATA32\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__data32</code> static and global variables in the <code>DATA32_I</code> segment. These values are copied from <code>DATA32_ID</code> to <code>DATA32_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

## DATA32\_N

|                     |                                                                    |
|---------------------|--------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __data32</code> variables. |
| Segment memory type | DATA                                                               |
| Memory placement    | This segment can be placed anywhere in memory.                     |
| Access type         | Read-write                                                         |

## DATA32\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__data32</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                           |

|                  |                                                |
|------------------|------------------------------------------------|
| Memory placement | This segment can be placed anywhere in memory. |
| Access type      | Read-write                                     |

## DIFUNCT

|                     |                                                                                                                                                             |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the dynamic initialization vector used by C++.                                                                                                        |
| Segment memory type | CONST                                                                                                                                                       |
| Memory placement    | In the Small data model, this segment must be placed in the first 64 Kbytes of memory. In other data models, this segment can be placed anywhere in memory. |
| Access type         | Read-only                                                                                                                                                   |

## IDINIT

|                     |                                                                            |
|---------------------|----------------------------------------------------------------------------|
| Description         | Holds information about the segments containing non-zero initialized data. |
| Segment memory type | CONST                                                                      |
| Memory placement    | This segment can be placed anywhere in memory.                             |
| Access type         | Read-only                                                                  |

## INDEX\_BASE

|                     |                                                                                                                               |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds an empty placeholder segment that is needed in front of the segments of the INDEX4, INDEX5, and INDEX20 segment groups. |
| Segment memory type | Any                                                                                                                           |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                |
| Access type         | Read-only                                                                                                                     |

## INDEX4\_I

|             |                                                                                                                                            |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__ix4</code> static and global initialized variables initialized by copying from the segment INDEX4_ID at application startup. |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------|



|                     |                                                                                                                                                                                                                                                                   |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                     | This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
| Segment memory type | DATA                                                                                                                                                                                                                                                              |
| Memory placement    | This segment must be placed within range 0–13 from the <code>INDEX_BASE</code> segment.                                                                                                                                                                           |
| Access type         | Read-write                                                                                                                                                                                                                                                        |

## INDEX4\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds initial values for <code>__ix4</code> static and global variables in the <code>INDEX4_I</code> segment. These values are copied from <code>INDEX4_ID</code> to <code>INDEX4_I</code> at application startup.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

## INDEX4\_N

|                     |                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __ix4</code> variables.                         |
| Segment memory type | DATA                                                                                    |
| Memory placement    | This segment must be placed within range 0–13 from the <code>INDEX_BASE</code> segment. |
| Access type         | Read-write                                                                              |

## INDEX4\_Z

|             |                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds zero-initialized <code>__ix4</code> static and global variables. The contents of this segment is declared by the system startup code. |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------|

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-z` directive must be used.

Segment memory type

DATA

Memory placement

This segment must be placed within range 0–13 from the `INDEX_BASE` segment.

Access type

Read-write

## INDEX5\_I

Description

Holds `__ix4` static and global initialized variables—scalars larger than a character and a structure without `char` members—initialized by copying from the segment `INDEX5_ID` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-z` directive must be used.

Segment memory type

DATA

Memory placement

This segment must be placed within range 0–27 from the `INDEX_BASE` segment.

Access type

Read-write

## INDEX5\_ID

Description

Holds initial values for `__ix4` static and global variables—scalars larger than a character and a structure without `char` members—in the `INDEX5_I` segment. These values are copied from `INDEX5_ID` to `INDEX5_I` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-z` directive must be used.

Segment memory type

CONST

Memory placement

This segment can be placed anywhere in memory.

Access type

Read-only

## INDEX5\_N

|                     |                                                                                                                                                   |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __ix4</code> variables—scalars larger than a character and a structure without <code>char</code> members. |
| Segment memory type | DATA                                                                                                                                              |
| Memory placement    | This segment must be placed within range 0–27 from the <code>INDEX_BASE</code> segment.                                                           |
| Access type         | Read-write                                                                                                                                        |

## INDEX5\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__ix4</code> static and global variables—scalars larger than a character and a structure without <code>char</code> members. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Memory placement    | This segment must be placed within range 0–27 from the <code>INDEX_BASE</code> segment.                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## INDEX20\_C

|                     |                                                                                                |
|---------------------|------------------------------------------------------------------------------------------------|
| Description         | Holds <code>__ix20</code> constant data, including string literals.                            |
| Segment memory type | CONST                                                                                          |
| Memory placement    | This segment must be placed within the range 1 Mbyte from the <code>INDEX_BASE</code> segment. |
| Access type         | Read-only                                                                                      |

## INDEX20\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__ix20</code> static and global initialized variables initialized by copying from the segment <code>INDEX20_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Memory placement    | This segment must be placed within the range 1 Mbyte from the <code>INDEX_BASE</code> segment.                                                                                                                                                                                                                                                                                                                                            |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                                                |

## INDEX20\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__ix20</code> static and global variables in the <code>INDEX20_I</code> segment. These values are copied from <code>INDEX20_ID</code> to <code>INDEX20_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

## INDEX20\_N

|                     |                                                                                                |
|---------------------|------------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __ix20</code> variables.                               |
| Segment memory type | DATA                                                                                           |
| Memory placement    | This segment must be placed within the range 1 Mbyte from the <code>INDEX_BASE</code> segment. |
| Access type         | Read-write                                                                                     |

## INDEX20\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__ix20</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Memory placement    | This segment must be placed within the range 1 Mbyte from the <code>INDEX_BASE</code> segment.                                                                                                                                                                                                                                                                                                                               |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                                   |

## INTVEC

|                     |                                                                                                                                                                       |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the interrupt vector table generated by the use of the <code>__interrupt</code> extended keyword in combination with the <code>#pragma vector</code> directive. |
| Segment memory type | CODE                                                                                                                                                                  |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                        |
| Access type         | Read-only                                                                                                                                                             |

## ISTACK

|             |                                                                                                                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds the stack used by interrupts and exceptions. This segment and its length are normally defined in the linker configuration file with this command:</p> <pre><code>-Z (DATA) ISTACK+nn=start</code></pre> |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

where *nn* is the size of the stack specified as a hexadecimal number and *start* is the first memory location.

|                     |                                                                                                                                                             |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Segment memory type | DATA                                                                                                                                                        |
| Memory placement    | In the Small data model, this segment must be placed in the first 64 Kbytes of memory. In other data models, this segment can be placed anywhere in memory. |
| Access type         | Read-write                                                                                                                                                  |

## ZINIT

|                     |                                                                            |
|---------------------|----------------------------------------------------------------------------|
| Description         | Holds information about the segments containing non-zero initialized data. |
| Segment memory type | CONST                                                                      |
| Memory placement    | This segment can be placed anywhere in memory.                             |
| Access type         | Read-only                                                                  |

# Implementation-defined behavior for Standard C

This chapter describes how the compiler handles the implementation-defined areas of the C language based on Standard C.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

If you are using C89 instead of Standard C, see *Implementation-defined behavior for C89*, page 327. For a short overview of the differences between Standard C and C89, see *C language overview*, page 145.

The text in this chapter applies to the DLIB library. Because the CLIB library does not follow Standard C, its implementation-defined behavior is not documented. For information about the CLIB library, see *The CLIB runtime environment*, page 113.

---

## Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

### J.3.1 TRANSLATION

#### Diagnostics (3.10, 5.1.1.3)

Diagnostics are produced in the form:

```
filename, linenumber level[tag]: message
```

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

**White-space characters (5.1.1.2)**

At translation phase three, each non-empty sequence of white-space characters is retained.

**J.3.2 ENVIRONMENT****The character set (5.1.1.2)**

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, if you use the `--enable_multibytes` compiler option, the host character set is used instead.

**Main (5.1.2.1)**

The function called at program startup is called `main`. No prototype is declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 95.

**The effect of program termination (5.1.2.1)**

Terminating the application returns the execution to the startup code (just after the call to `main`).

**Alternative ways to define main (5.1.2.2.1)**

There is no alternative ways to define the `main` function.

**The argv argument to main (5.1.2.2.1)**

The `argv` argument is not supported.

**Streams as interactive devices (5.1.2.3)**

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

**Signals, their semantics, and the default handling (7.14)**

In the DLIB library, the set of supported signals is the same as in Standard C. A raised signal will do nothing, unless the `signal` function is customized to fit the application.



**Signal values for computational exceptions (7.14.1.1)**

In the DLIB library, there are no implementation-defined values that correspond to a computational exception.

**Signals at system startup (7.14.1.1)**

In the DLIB library, there are no implementation-defined signals that are executed at system startup.

**Environment names (7.20.4.5)**

In the DLIB library, there are no implementation-defined environment names that are used by the `getenv` function.

**The system function (7.20.4.6)**

The `system` function is not supported.

**J.3.3 IDENTIFIERS****Multibyte characters in identifiers (6.4.2)**

Additional multibyte characters may not appear in identifiers.

**Significant characters in identifiers (5.2.4.1, 6.1.2)**

The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

**J.3.4 CHARACTERS****Number of bits in a byte (3.6)**

A byte contains 8 bits.

**Execution character set member values (5.2.1)**

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the host character set.

**Alphabetic escape sequences (5.2.2)**

The standard alphabetic escape sequences have the values `\a-7`, `\b-8`, `\f-12`, `\n-10`, `\r-13`, `\t-9`, and `\v-11`.

### **Characters outside of the basic executive character set (6.2.5)**

A character outside of the basic executive character set that is stored in a `char` is not transformed.

### **Plain char (6.2.5, 6.3.1.1)**

A plain `char` is treated as an `unsigned char`.

### **Source and execution character sets (6.4.4.4, 5.1.1.2)**

The source character set is the set of legal characters that can appear in source files. By default, the source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. By default, the execution character set is the standard ASCII character set.

However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 102.

### **Integer character constants with more than one character (6.4.4.4)**

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

### **Wide character constants with more than one character (6.4.4.4)**

A wide character constant that contains more than one multibyte character generates a diagnostic message.

### **Locale used for wide character constants (6.4.4.4)**

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

### **Locale used for wide string literals (6.4.5)**

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

**Source characters as executive characters (6.4.5)**

All source characters can be represented as executive characters.

**J.3.5 INTEGERS****Extended integer types (6.2.5)**

There are no extended integer types.

**Range of integer values (6.2.6.2)**

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

For information about the ranges for the different integer types, see *Basic data types*, page 228.

**The rank of extended integer types (6.3.1.1)**

There are no extended integer types.

**Signals when converting to a signed integer type (6.3.1.3)**

No signal is raised when an integer is converted to a signed integer type.

**Signed bitwise operations (6.5)**

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

**J.3.6 FLOATING POINT****Accuracy of floating-point operations (5.2.4.2.2)**

The accuracy of floating-point operations is unknown.

**Rounding behaviors (5.2.4.2.2)**

There are no non-standard values of `FLT_ROUNDS`.

**Evaluation methods (5.2.4.2.2)**

There are no non-standard values of `FLT_EVAL_METHOD`.

### **Converting integer values to floating-point values (6.3.1.4)**

When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Converting floating-point values to floating-point values (6.3.1.5)**

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Denoting the value of floating-point constants (6.4.4.2)**

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Contraction of floating-point values (6.5)**

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

### **Default state of `FENV_ACCESS` (7.6.1)**

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

### **Additional floating-point mechanisms (7.6, 7.12)**

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

### **Default state of `FP_CONTRACT` (7.12.2)**

The default state of the pragma directive `FP_CONTRACT` is `OFF`.

## **J.3.7 ARRAYS AND POINTERS**

### **Conversion from/to pointers (6.3.2.3)**

For information about casting of data pointers and function pointers, see *Casting*, page 233.

### **`ptrdiff_t` (6.5.6)**

For information about `ptrdiff_t`, see *ptrdiff\_t*, page 234.

### J.3.8 HINTS

#### Honoring the register keyword (6.7.1)

User requests for register variables are not honored.

#### Inlining functions (6.7.4)

User requests for inlining functions increases the chance, but does not make it certain, that the function will actually be inlined into another function. See *Inlining functions*, page 63.

### J.3.9 STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

#### Sign of 'plain' bitfields (6.7.2, 6.7.2.1)

For information about how a 'plain' `int` bitfield is treated, see *Bitfields*, page 229.

#### Possible types for bitfields (6.7.2.1)

All integer types can be used as bitfields in the compiler's extended mode, see *-e*, page 208.

#### Bitfields straddling a storage-unit boundary (6.7.2.1)

A bitfield is always placed in one—and one only—storage unit, which means that the bitfield cannot straddle a storage-unit boundary.

#### Allocation order of bitfields within a unit (6.7.2.1)

For information about how bitfields are allocated within a storage unit, see *Bitfields*, page 229.

#### Alignment of non-bitfield structure members (6.7.2.1)

The alignment of non-bitfield members of structures is the same as for the member types, see *Alignment*, page 227.

#### Integer type used for representing enumeration types (6.7.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

### J.3.10 QUALIFIERS

#### Access to volatile objects (6.7.3)

Any reference to an object with `volatile` qualified type is an access, see *Declaring objects volatile*, page 236.

### J.3.11 PREPROCESSING DIRECTIVES

#### Mapping of header names (6.4.7)

Sequences in header names are mapped to source file names verbatim. A backslash `'\'` is not treated as an escape sequence. See *Overview of the preprocessor*, page 277.

#### Character constants in constant expressions (6.10.1)

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

#### The value of a single-character constant (6.10.1)

A single-character constant may only have a negative value if a plain character (`char`) is treated as a signed character, see *--char\_is\_signed*, page 201.

#### Including bracketed filenames (6.10.2)

For information about the search algorithm used for file specifications in angle brackets `<>`, see *Include file search procedure*, page 190.

#### Including quoted filenames (6.10.2)

For information about the search algorithm used for file specifications enclosed in quotes, see *Include file search procedure*, page 190.

#### Preprocessing tokens in #include directives (6.10.2)

Preprocessing tokens in an `#include` directive are combined in the same way as outside an `#include` directive.

#### Nesting limits for #include directives (6.10.2)

There is no explicit nesting limit for `#include` processing.

#### Universal character names (6.10.3.2)

Universal character names (UCN) are not supported.

### Recognized pragma directives (6.10.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter overrides the information here.

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
constseg
cspy_support
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_default_requirements
library_provides
library_requirement_override
memory
module_name
no_pch
once
public_equ
system_include
```

warnings

### **Default `__DATE__` and `__TIME__` (6.10.8)**

The definitions for `__TIME__` and `__DATE__` are always available.

## **J.3.12 LIBRARY FUNCTIONS**

### **Additional library facilities (5.1.2.1)**

Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 79.

### **Diagnostic printed by the `assert` function (7.2.1.1)**

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

### **Representation of the floating-point status flags (7.6.2.2)**

For information about the floating-point status flags, see *fenv.h*, page 289.

### **`feraiseexcept` raising floating-point exception (7.6.2.3)**

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 231.

### **Strings passed to the `setlocale` function (7.11.1.1)**

For information about strings passed to the `setlocale` function, see *Locale*, page 102.

### **Types defined for `float_t` and `double_t` (7.12)**

The `FLT_EVAL_METHOD` macro can only have the value 0.

### **Domain errors (7.12.1)**

No function generates other domain errors than what the standard requires.

### **Return values on domain errors (7.12.1)**

Mathematic functions return a floating-point NaN (not a number) for domain errors.



**Underflow errors (7.12.1)**

Mathematic functions set `errno` to the macro `ERANGE` (a macro in `errno.h`) and return zero for underflow errors.

**fmod return value (7.12.10.1)**

The `fmod` function returns a floating-point NaN when the second argument is zero.

**The magnitude of remquo (7.12.10.3)**

The magnitude is congruent modulo `INT_MAX`.

**signal() (7.14.1.1)**

The signal part of the library is not supported.

**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 106.

**NULL macro (7.17)**

The `NULL` macro is defined to 0.

**Terminating newline character (7.19.2)**

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

**Space characters before a newline character (7.19.2)**

Space characters written to a stream immediately before a newline character are preserved.

**Null characters appended to data written to binary streams (7.19.2)**

No null characters are appended to data written to binary streams.

**File position in append mode (7.19.3)**

The file position is initially placed at the beginning of the file when it is opened in append-mode.

### **Truncation of files (7.19.3)**

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 102.

### **File buffering (7.19.3)**

An open file can be either block-buffered, line-buffered, or unbuffered.

### **A zero-length file (7.19.3)**

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

### **Legal file names (7.19.3)**

The legality of a filename depends on the application-specific implementation of the low-level file routines.

### **Number of times a file can be opened (7.19.3)**

Whether a file can be opened more than once depends on the application-specific implementation of the low-level file routines.

### **Multibyte characters in a file (7.19.3)**

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

### **remove() (7.19.4.1)**

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 102.

### **rename() (7.19.4.2)**

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 102.

### **Removal of open temporary files (7.19.4.3)**

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

**Mode changing (7.19.5.4)**

`freopen` closes the named stream, then reopens it in the new mode. The streams `stdin`, `stdout`, and `stderr` can be reopened in any new mode.

**Style for printing infinity or NaN (7.19.6.1, 7.24.2.1)**

The style used for printing infinity or NaN for a floating-point constant is `inf` and `nan` (`INF` and `NAN` for the `F` conversion specifier), respectively. The `n`-char-sequence is not used for `nan`.

**%p in printf() (7.19.6.1, 7.24.2.1)**

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

**Reading ranges in scanf (7.19.6.2, 7.24.2.1)**

A `-` (dash) character is always treated as a range symbol.

**%p in scanf (7.19.6.2, 7.24.2.2)**

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

**File position errors (7.19.9.1, 7.19.9.3, 7.19.9.4)**

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

**An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)**

An `n`-char-sequence after a NaN is read and ignored.

**errno value at underflow (7.20.1.3, 7.24.4.1.1)**

`errno` is set to `ERANGE` if an underflow is encountered.

**Zero-sized heap objects (7.20.3)**

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

**Behavior of abort and exit (7.20.4.1, 7.20.4.4)**

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

### Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)

The termination status will be propagated to `__exit()` as a parameter. `exit()` and `_Exit()` use the input parameter, whereas `abort` uses `EXIT_FAILURE`.

### The system function return value (7.20.4.6)

The `system` function is not supported.

### The time zone (7.23.1)

The local time zone and daylight savings time must be defined by the application. For more information, see *Time*, page 106.

### Range and precision of time (7.23)

The implementation uses `signed long` for representing `clock_t` and `time_t`, based at the start of the year 1970. This gives a range of approximately plus or minus 69 years in seconds. However, the application must supply the actual implementation for the functions `time` and `clock`. See *Time*, page 106.

### clock() (7.23.2.1)

The application must supply an implementation of the `clock` function. See *Time*, page 106.

### %Z replacement string (7.23.3.5, 7.24.5.1)

By default, ":" is used as a replacement for %Z. Your application should implement the time zone handling. See *Time*, page 106.

### Math functions rounding mode (F.9)

The functions in `math.h` honor the rounding direction mode in `FLT-ROUNDS`.

## J.3.13 ARCHITECTURE

### Values and expressions assigned to some macros (5.2.4.2, 7.18.2, 7.18.3)

There are always 8 bits in a byte.

`MB_LEN_MAX` is at the most 6 bytes depending on the library configuration that is used.

For information about sizes, ranges, etc for all basic types, see *Data representation*, page 227.

The limit macros for the exact-width, minimum-width, and fastest minimum-width integer types defined in `stdint.h` have the same ranges as `char`, `short`, `int`, `long`, and `long long`.

The floating-point constant `FLT_ROUNDS` has the value 1 (to nearest) and the floating-point constant `FLT_EVAL_METHOD` has the value 0 (treat as is).

### **The number, order, and encoding of bytes (6.2.6.1)**

See *Data representation*, page 227.

### **The value of the result of the sizeof operator (6.5.3.4)**

See *Data representation*, page 227.

## **J.4 LOCALE**

### **Members of the source and execution character set (5.2.1)**

By default, the compiler accepts all one-byte characters in the host's default character set. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

#### **The meaning of the additional character set (5.2.1.2)**

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

#### **Shift states for encoding multibyte characters (5.2.1.2)**

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

#### **Direction of successive printing characters (5.2.2)**

The application defines the characteristics of a display device.

### **The decimal point character (7.1.1)**

The default decimal-point character is a `'.'`. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

### **Printing characters (7.4, 7.25.2)**

The set of printing characters is determined by the chosen locale.

### Control characters (7.4, 7.25.2)

The set of control characters is determined by the chosen locale.

### Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

The sets of characters tested are determined by the chosen locale.

### The native environment (7.1.1.1)

The native environment is the same as the "C" locale.

### Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

### The collation of the execution character set (7.21.4.3, 7.24.4.4.2)

The collation of the execution character set is determined by the chosen locale.

### Message returned by strerror (7.21.6.2)

The messages returned by the `strerror` function depending on the argument is:

| Argument   | Message                   |
|------------|---------------------------|
| EZERO      | no error                  |
| EDOM       | domain error              |
| ERANGE     | range error               |
| EFPOS      | file positioning error    |
| EILSEQ     | multi-byte encoding error |
| <0    >99  | unknown error             |
| all others | error <i>nnn</i>          |

Table 43: Message returned by `strerror()`—IAR DLIB library

# Implementation-defined behavior for C89

This chapter describes how the compiler handles the implementation-defined areas of the C language based on the C89 standard.

If you are using Standard C instead of C89, see *Implementation-defined behavior for Standard C*, page 311. For a short overview of the differences between Standard C and C89, see *C language overview*, page 145.

---

## Descriptions of implementation-defined behavior

The descriptions follow the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

### TRANSLATION

#### Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

```
filename, linenumber level[tag): message
```

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

### ENVIRONMENT

#### Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 95. To change this behavior for the IAR CLIB runtime environment, see *Customizing system initialization*, page 119.

### **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. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 102.

### **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. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 102.

### 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 228, 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 231, 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 234, for information about *size\_t*.

### **Conversion from/to pointers (6.3.4)**

See *Casting*, page 233, for information about casting of data pointers and function pointers.

### **ptrdiff\_t (6.3.6, 7.1.1)**

See *ptrdiff\_t*, page 234, 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 228, 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 signed *int* bitfield. All integer types are allowed as bitfields.

### **Allocation order of bitfields within a unit (6.5.2.1)**

Bitfields are allocated within an integer from least-significant to most-significant bit.

### **Can bitfields straddle a storage-unit boundary (6.5.2.1)**

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

### **Integer type chosen to represent enumeration types (6.5.2.2)**

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

## **QUALIFIERS**

### **Access to volatile objects (6.5.3)**

Any reference to an object with volatile qualified type is an access.

## **DECLARATORS**

### **Maximum numbers of declarators (6.5.4)**

The number of declarators is not limited. The number is limited only by the available memory.

## **STATEMENTS**

### **Maximum number of case statements (6.6.4.2)**

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

## **PREPROCESSING DIRECTIVES**

### **Character constants and conditional inclusion (6.8.1)**

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

### **Including bracketed filenames (6.8.2)**

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

### **Including quoted filenames (6.8.2)**

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source

file currently being processed. If there is no grandparent file and the file is not found, the search continues as if the filename was enclosed in angle brackets.

### Character sequences (6.8.2)

Preprocessor directives use the source character set, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

### Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter overrides the information here.

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
constseg
cspy_support
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_default_requirements
```

```

library_provides
library_requirement_override
memory
module_name
no_pch
once
public_equ
system_include
warnings

```

### **Default `__DATE__` and `__TIME__` (6.8.8)**

The definitions for `__TIME__` and `__DATE__` are always available.

## **IAR DLIB LIBRARY FUNCTIONS**

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

### **NULL macro (7.1.6)**

The `NULL` macro is defined to 0.

### **Diagnostic printed by the `assert` function (7.2)**

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

### **Domain errors (7.5.1)**

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

### **Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)**

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

**fmod() functionality (7.5.6.4)**

If the second argument to `fmod()` is zero, the function returns NaN; `errno` is set to EDOM.

**signal() (7.7.1.1)**

The signal part of the library is not supported.

**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 106.

**Terminating newline character (7.9.2)**

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

**Blank lines (7.9.2)**

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

**Null characters appended to data written to binary streams (7.9.2)**

No null characters are appended to data written to binary streams.

**Files (7.9.3)**

Whether the file position indicator of an append-mode stream is initially positioned at the beginning or the end of the file, depends on the application-specific implementation of the low-level file routines.

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 102.

The characteristics of the file buffering is that the implementation supports files that are unbuffered, line buffered, or fully buffered.

Whether a zero-length file actually exists depends on the application-specific implementation of the low-level file routines.

Rules for composing valid file names depends on the application-specific implementation of the low-level file routines.

Whether the same file can be simultaneously open multiple times depends on the application-specific implementation of the low-level file routines.

**remove() (7.9.4.1)**

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 102.

**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 102.

**%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 105.

### `system()` (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 105.

### Message returned by `strerror()` (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

| Argument   | Message                   |
|------------|---------------------------|
| EZERO      | no error                  |
| EDOM       | domain error              |
| ERANGE     | range error               |
| EFPOS      | file positioning error    |
| EILSEQ     | multi-byte encoding error |
| <0    >99  | unknown error             |
| all others | error <i>nnn</i>          |

Table 44: 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 106.

### `clock()` (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 106.

## IAR CLIB LIBRARY FUNCTIONS

### NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0`.

### **Diagnostic printed by the assert function (7.2)**

The `assert()` function prints:

Assertion failed: *expression*, file *Filename*, line *linenumber*

when the parameter evaluates to zero.

### **Domain errors (7.5.1)**

`HUGE_VAL`, the largest representable value in a double floating-point type, 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 zero (it does not change the integer expression `errno`).

### **`signal()` (7.7.1.1)**

The signal part of the library is not supported.

### **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)**

There are no binary streams implemented.

### **Files (7.9.3)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**remove() (7.9.4.1)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**rename() (7.9.4.2)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**%p in printf() (7.9.6.1)**

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. 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 explicitly as a `-` character.

**File position errors (7.9.9.1, 7.9.9.4)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**Message generated by perror() (7.9.10.4)**

`perror()` is not supported.

**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 `exit()` function does not return.

### Environment (7.10.4.4)

Environments are not supported.

### system() (7.10.4.5)

The `system()` function is not supported.

### Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument are:

| Argument   | Message       |
|------------|---------------|
| EZERO      | no error      |
| EDOM       | domain error  |
| ERANGE     | range error   |
| <0    >99  | unknown error |
| all others | error No.xx   |

*Table 45: Message returned by strerror()—IAR CLIB library*

### The time zone (7.12.1)

The time zone function is not supported.

### clock() (7.12.2.1)

The `clock()` function is not supported.

## A

- abort
  - implementation-defined behavior. . . . . 323
  - implementation-defined behavior in C89 (CLIB). . . . . 339
  - implementation-defined behavior in C89 (DLIB) . . . . . 336
  - system termination (DLIB) . . . . . 95
- absolute location
  - data, placing at (@) . . . . . 174
  - language support for . . . . . 148
  - #pragma location . . . . . 258
- ABSOLUTE\_C (segment) . . . . . 295
- ABSOLUTE\_N (segment) . . . . . 295
- addressing. *See* memory types, data models, and code models
- \_\_adjust\_return\_address (intrinsic function). . . . . 270
- algorithm (STL header file) . . . . . 287
- alignment . . . . . 227
  - forcing stricter (#pragma data\_alignment). . . . . 253
  - in structures (#pragma pack) . . . . . 261
  - in structures, causing problems . . . . . 171
  - of an object (\_\_ALIGNOF\_\_) . . . . . 148
  - of data types. . . . . 227
  - restrictions for inline assembler . . . . . 124
- alignment (pragma directive) . . . . . 319, 333
- \_\_ALIGNOF\_\_ (operator) . . . . . 148
- anonymous structures . . . . . 171
- anonymous symbols, creating . . . . . 145
- ANSI C. *See* C89
- application
  - building, overview of . . . . . 36
  - startup and termination (CLIB) . . . . . 117
  - startup and termination (DLIB) . . . . . 92
- architecture
  - more information about . . . . . 25
  - of CR16C. . . . . 43
- ARGFRAME (assembler directive) . . . . . 135
- argv (argument), implementation-defined behavior . . . . . 312
- arrays
  - designated initializers in . . . . . 145
  - global, accessing . . . . . 136
  - hints about index type . . . . . 170
  - implementation-defined behavior. . . . . 316
  - implementation-defined behavior in C89 . . . . . 331
  - incomplete at end of structs . . . . . 145
  - non-lvalue . . . . . 151
  - of incomplete types . . . . . 150
  - single-value initialization. . . . . 151
- asm, \_\_asm (language extension) . . . . . 123
- assembler code
  - calling from C . . . . . 125
  - calling from C++ . . . . . 127
  - inserting inline . . . . . 123
- assembler directives
  - for call frame information . . . . . 139
  - for static overlay . . . . . 134
  - using in inline assembler code . . . . . 124
- assembler instructions
  - inserting inline . . . . . 123
  - JUSR . . . . . 249
  - MACQW . . . . . 272
  - MACSW . . . . . 273
  - MACUW . . . . . 273
  - used for calling functions. . . . . 135
- assembler labels, making public (--public\_equ) . . . . . 220
- assembler language interface . . . . . 121
  - calling convention. *See* assembler code
- assembler list file, generating . . . . . 211
- assembler output file. . . . . 126
- asserts . . . . . 109
  - implementation-defined behavior of . . . . . 320
  - implementation-defined behavior of in C89, (CLIB) . . . . . 338
  - implementation-defined behavior of in C89, (DLIB) . . . . . 334
  - including in application . . . . . 282
- assert.h (CLIB header file) . . . . . 291
- assert.h (DLIB header file) . . . . . 285
- \_\_assignment\_by\_bitwise\_copy\_allowed, symbol used in library . . . . . 290

|                                      |     |
|--------------------------------------|-----|
| @ (operator)                         |     |
| placing at absolute address          | 174 |
| placing in segments                  | 175 |
| atomic operations                    | 58  |
| __monitor                            | 247 |
| attributes                           |     |
| object                               | 242 |
| type                                 | 239 |
| auto variables                       | 52  |
| at function entrance                 | 130 |
| programming hints for efficient code | 182 |
| using in inline assembler code       | 124 |

## B

|                                                 |                                   |
|-------------------------------------------------|-----------------------------------|
| backtrace information                           | <i>See</i> call frame information |
| Barr, Michael                                   | 28                                |
| baseaddr (pragma directive)                     | 319, 333                          |
| __BASE_FILE__ (predefined symbol)               | 278                               |
| basic_template_matching (pragma directive)      | 252                               |
| using                                           | 163                               |
| batch files                                     |                                   |
| error return codes                              | 192                               |
| none for building library from command line     | 91                                |
| binary streams                                  | 321                               |
| binary streams in C89 (CLIB)                    | 338                               |
| binary streams in C89 (DLIB)                    | 335                               |
| bit negation                                    | 183                               |
| bitfields                                       |                                   |
| data representation of                          | 229                               |
| hints                                           | 169                               |
| implementation-defined behavior                 | 317                               |
| implementation-defined behavior in C89          | 331                               |
| non-standard types in                           | 148                               |
| bitfields (pragma directive)                    | 253                               |
| bits in a byte, implementation-defined behavior | 313                               |
| bold style, in this guide                       | 30                                |
| bool (data type)                                | 228                               |
| adding support for in CLIB                      | 291                               |

|                                      |          |
|--------------------------------------|----------|
| adding support for in DLIB           | 286, 288 |
| building_runtime (pragma directive)  | 319, 333 |
| __BUILD_NUMBER__ (predefined symbol) | 278      |
| byte order, identifying              | 280      |

## C

|                                               |                               |
|-----------------------------------------------|-------------------------------|
| C and C++ linkage                             | 128                           |
| C/C++ calling convention                      | <i>See</i> calling convention |
| C header files                                | 285                           |
| C language, overview                          | 145                           |
| call frame information                        | 139                           |
| in assembler list file                        | 126                           |
| in assembler list file (-IA)                  | 211                           |
| call stack                                    | 139                           |
| callee-save registers, stored on stack        | 52                            |
| calling convention                            |                               |
| C++, requiring C linkage                      | 127                           |
| in compiler                                   | 128                           |
| calloc (library function)                     | 53                            |
| <i>See also</i> heap                          |                               |
| implementation-defined behavior in C89 (CLIB) | 339                           |
| implementation-defined behavior in C89 (DLIB) | 336                           |
| can_instantiate (pragma directive)            | 319, 333                      |
| cassert (library header file)                 | 288                           |
| cast operators                                |                               |
| in Extended EC++                              | 154, 165                      |
| missing from Embedded C++                     | 154                           |
| casting                                       |                               |
| of pointers and integers                      | 233                           |
| pointers to integers, language extension      | 150                           |
| cctype (DLIB header file)                     | 288                           |
| cerrno (DLIB header file)                     | 288                           |
| cexit (system termination code)               |                               |
| in CLIB                                       | 117                           |
| placement in segment                          | 75                            |
| CFI (assembler directive)                     | 139                           |
| CFI_COMMON (call frame information macro)     | 143                           |
| CFI_NAMES (call frame information macro)      | 143                           |

- cfi.m45 (CFI header example file) . . . . . 143
- cfloat (DLIB header file) . . . . . 288
- char (data type) . . . . . 228
  - changing default representation (`--char_is_signed`) . . . 201
  - changing representation (`--char_is_unsigned`) . . . . . 201
  - implementation-defined behavior . . . . . 314
  - signed and unsigned . . . . . 229
- character set, implementation-defined behavior . . . . . 312
- characters, implementation-defined behavior . . . . . 313
- characters, implementation-defined behavior in C89 . . . 328
- character-based I/O
  - in CLIB . . . . . 115
  - in DLIB . . . . . 98
- `--char_is_signed` (compiler option) . . . . . 201
- `--char_is_unsigned` (compiler option) . . . . . 201
- CHECKSUM (segment) . . . . . 296
- cinttypes (DLIB header file) . . . . . 288
- class memory (extended EC++) . . . . . 156
- class template partial specialization
  - matching (extended EC++) . . . . . 162
- `__clear_processor_register_bit` (intrinsic function) . . . . . 270
- `__clear_PSR_I_bit` (intrinsic function) . . . . . 271
- CLIB . . . . . 39, 291
  - documentation . . . . . 28
  - runtime environment . . . . . 113
  - summary of definitions . . . . . 291
- `--clib` (compiler option) . . . . . 201
- climits (DLIB header file) . . . . . 288
- locale (DLIB header file) . . . . . 288
- clock (CLIB library function),
  - implementation-defined behavior in C89 . . . . . 340
- clock (DLIB library function),
  - implementation-defined behavior in C89 . . . . . 337
- clock (library function)
  - implementation-defined behavior . . . . . 324
- clock.c . . . . . 106
- `__close` (DLIB library function) . . . . . 102
- cmath (DLIB header file) . . . . . 288
- code
  - execution of . . . . . 38
  - interruption of execution . . . . . 56
  - verifying linked result . . . . . 76
- code models . . . . . 55
  - calling functions in . . . . . 135
  - configuration . . . . . 38
  - identifying (`__CODE_MODEL__`) . . . . . 278
  - specifying on command line (`--code_model`) . . . . . 202
- code motion (compiler transformation) . . . . . 180
  - disabling (`--no_code_motion`) . . . . . 213
- code pointers . . . . . 233
- code segments, used for placement . . . . . 75
- CODE (segment) . . . . . 296
  - using . . . . . 76
- codeseg (pragma directive) . . . . . 319, 333
- `__CODE_MODEL__` (predefined symbol) . . . . . 278
- `__code_model` (runtime model attribute) . . . . . 111
- `--code_model` (compiler option) . . . . . 202
- `__code`, symbol used in library . . . . . 290
- command line options
  - See also* compiler options
  - part of compiler invocation syntax . . . . . 189
  - passing . . . . . 189
  - typographic convention . . . . . 30
- command prompt icon, in this guide . . . . . 30
- comments
  - after preprocessor directives . . . . . 151
  - C++ style, using in C code . . . . . 145
- common block (call frame information) . . . . . 140
- common subexpr elimination (compiler transformation) . 179
  - disabling (`--no_cse`) . . . . . 214
- compilation date
  - exact time of (`__TIME__`) . . . . . 281
  - identifying (`__DATE__`) . . . . . 279
- compiler
  - environment variables . . . . . 190
  - invocation syntax . . . . . 189
  - output from . . . . . 191
- compiler listing, generating (-l) . . . . . 211

|                                                        |        |                                                |          |
|--------------------------------------------------------|--------|------------------------------------------------|----------|
| compiler object file . . . . .                         | 36     | constseg (pragma directive) . . . . .          | 319, 333 |
| including debug information in (--debug, -r) . . . . . | 203    | const_cast (cast operator) . . . . .           | 154      |
| output from compiler . . . . .                         | 191    | contents, of this guide . . . . .              | 25       |
| compiler optimization levels . . . . .                 | 178    | control characters,                            |          |
| compiler options . . . . .                             | 195    | implementation-defined behavior . . . . .      | 325      |
| passing to compiler . . . . .                          | 189    | conventions, used in this guide . . . . .      | 29       |
| reading from file (-f) . . . . .                       | 210    | copyright notice . . . . .                     | 2        |
| specifying parameters . . . . .                        | 197    | cos (library function) . . . . .               | 284      |
| summary . . . . .                                      | 197    | cos (library routine) . . . . .                | 107–108  |
| syntax . . . . .                                       | 195    | cosf (library routine) . . . . .               | 108–109  |
| for creating skeleton code . . . . .                   | 126    | cosl (library routine) . . . . .               | 108–109  |
| --warnings_affect_exit_code . . . . .                  | 192    | __cplusplus (predefined symbol) . . . . .      | 278      |
| compiler platform, identifying . . . . .               | 280    | cross call (compiler transformation) . . . . . | 181      |
| compiler subversion number . . . . .                   | 281    | CR16C                                          |          |
| compiler transformations . . . . .                     | 177    | instruction set . . . . .                      | 136      |
| compiler version number . . . . .                      | 281    | memory access . . . . .                        | 38       |
| compiling                                              |        | memory layout . . . . .                        | 43       |
| from the command line . . . . .                        | 36     | csetjmp (DLIB header file) . . . . .           | 288      |
| syntax . . . . .                                       | 189    | csignal (DLIB header file) . . . . .           | 288      |
| complex numbers, supported in Embedded C++. . . . .    | 154    | csy_support (pragma directive) . . . . .       | 319, 333 |
| complex (library header file) . . . . .                | 287    | CSTACK (segment) . . . . .                     | 296      |
| complex.h (library header file) . . . . .              | 285    | example . . . . .                              | 72       |
| compound literals . . . . .                            | 145    | <i>See also</i> stack                          |          |
| computer style, typographic convention . . . . .       | 30     | CSTART (segment) . . . . .                     | 75, 296  |
| configuration                                          |        | cstartup (system startup code)                 |          |
| basic project settings . . . . .                       | 37     | code segment for . . . . .                     | 75       |
| __low_level_init . . . . .                             | 96     | customizing system initialization . . . . .    | 95       |
| configuration symbols                                  |        | source files for (CLIB) . . . . .              | 117      |
| for file input and output . . . . .                    | 102    | source files for (DLIB) . . . . .              | 92       |
| for locale . . . . .                                   | 103    | cstdarg (DLIB header file) . . . . .           | 288      |
| for printf and scanf . . . . .                         | 100    | cstdbool (DLIB header file) . . . . .          | 288      |
| for strtod . . . . .                                   | 107    | cstddef (DLIB header file) . . . . .           | 288      |
| in library configuration files . . . . .               | 92, 96 | cstdlibio (DLIB header file) . . . . .         | 288      |
| consistency, module . . . . .                          | 110    | cstdliblib (DLIB header file) . . . . .        | 288      |
| const                                                  |        | cstring (DLIB header file) . . . . .           | 288      |
| declaring objects . . . . .                            | 238    | ctime (DLIB header file) . . . . .             | 288      |
| non-top level . . . . .                                | 151    | ctype.h (library header file) . . . . .        | 286, 291 |
| __constrange(), symbol used in library . . . . .       | 290    | cwctype.h (library header file) . . . . .      | 289      |
| __construction_by_bitwise_copy_allowed, symbol used    |        | ?C_EXIT (assembler label) . . . . .            | 119      |
| in library . . . . .                                   | 290    | ?C_GETCHAR (assembler label) . . . . .         | 119      |



- C\_INCLUDE (environment variable) . . . . . 190
  - ?C\_PUTCHAR (assembler label) . . . . . 119
  - C-SPY
    - debug support for C++ . . . . . 161
    - including debugging support . . . . . 86
    - interface to system termination . . . . . 95
    - low-level interface (CLIB) . . . . . 119
    - Terminal I/O window, including debug support for . . . . . 88
  - C++
    - See also* Embedded C++ and Extended Embedded C++
    - absolute location . . . . . 175–176
    - calling convention . . . . . 127
    - dynamic initialization in . . . . . 76
    - header files . . . . . 286
    - language extensions . . . . . 166
    - special function types . . . . . 62
    - standard template library (STL) . . . . . 287
    - static member variables . . . . . 175–176
    - support for . . . . . 35
  - C++ header files . . . . . 287
  - C++ names, in assembler code . . . . . 127
  - C++ objects, placing in memory type . . . . . 51
  - C++ terminology . . . . . 29
  - C++-style comments . . . . . 145
  - C89
    - implementation-defined behavior . . . . . 327
    - support for . . . . . 145
  - c89 (compiler option) . . . . . 200
  - C99. *See* Standard C
- ## D
- D (compiler option) . . . . . 202
  - data
    - alignment of . . . . . 227
    - different ways of storing . . . . . 43
    - located, declaring extern . . . . . 175
    - placing . . . . . 173, 222, 293
    - at absolute location . . . . . 174
    - representation of . . . . . 227
    - storage . . . . . 43
    - verifying linked result . . . . . 76
  - data block (call frame information) . . . . . 140
  - data memory attributes, using . . . . . 47
  - data models . . . . . 44
    - configuration . . . . . 38
    - huge . . . . . 45
    - identifying (`__DATA_MODEL__`) . . . . . 278
    - indexed . . . . . 45
    - large . . . . . 45
    - medium . . . . . 45
    - small . . . . . 45
  - data pointers . . . . . 233
  - data segments . . . . . 69
  - data types . . . . . 228
    - avoiding signed . . . . . 169
    - floating point . . . . . 231
    - in C++ . . . . . 238
    - integer types . . . . . 228
  - dataseg (pragma directive) . . . . . 319, 333
  - data\_alignment (pragma directive) . . . . . 253
  - `__DATA_MODEL__` (predefined symbol) . . . . . 278
  - data\_model (compiler option) . . . . . 203
  - `__data16` (extended keyword) . . . . . 243
  - data16 (memory type) . . . . . 46
  - DATA16\_C (segment) . . . . . 297
  - DATA16\_HEAP (segment) . . . . . 74, 297
  - DATA16\_I (segment) . . . . . 297
  - DATA16\_ID (segment) . . . . . 298
  - DATA16\_N (segment) . . . . . 298
  - DATA16\_Z (segment) . . . . . 298
  - `__data20` (extended keyword) . . . . . 244
  - data20 (memory type) . . . . . 47
  - DATA20\_C (segment) . . . . . 299
  - DATA20\_I (segment) . . . . . 299
  - DATA20\_ID (segment) . . . . . 299
  - DATA20\_N (segment) . . . . . 300
  - DATA20\_Z (segment) . . . . . 300

|                                                               |          |                                                                    |         |
|---------------------------------------------------------------|----------|--------------------------------------------------------------------|---------|
| __data24 (extended keyword) . . . . .                         | 244      | classifying as compiler warnings . . . . .                         | 206     |
| data24 (memory type) . . . . .                                | 47       | disabling compiler warnings . . . . .                              | 217     |
| DATA24_C (segment) . . . . .                                  | 300      | disabling wrapping of in compiler . . . . .                        | 217     |
| DATA24_I (segment) . . . . .                                  | 300      | enabling compiler remarks . . . . .                                | 221     |
| DATA24_ID (segment) . . . . .                                 | 301      | listing all used by compiler . . . . .                             | 206     |
| DATA24_N (segment) . . . . .                                  | 301      | suppressing in compiler . . . . .                                  | 205     |
| DATA24_Z (segment) . . . . .                                  | 301      | --diagnostics_tables (compiler option) . . . . .                   | 206     |
| __data32 (extended keyword) . . . . .                         | 245      | diagnostics, implementation-defined behavior . . . . .             | 311     |
| data32 (memory type) . . . . .                                | 47       | diag_default (pragma directive) . . . . .                          | 254     |
| DATA32_C (segment) . . . . .                                  | 302      | --diag_error (compiler option) . . . . .                           | 204     |
| DATA32_HEAP (segment) . . . . .                               | 74, 302  | diag_error (pragma directive) . . . . .                            | 254     |
| DATA32_I (segment) . . . . .                                  | 302      | --diag_remark (compiler option) . . . . .                          | 205     |
| DATA32_ID (segment) . . . . .                                 | 303      | diag_remark (pragma directive) . . . . .                           | 255     |
| DATA32_N (segment) . . . . .                                  | 303      | --diag_suppress (compiler option) . . . . .                        | 205     |
| DATA32_Z (segment) . . . . .                                  | 303      | diag_suppress (pragma directive) . . . . .                         | 255     |
| __DATE__ (predefined symbol) . . . . .                        | 279      | --diag_warning (compiler option) . . . . .                         | 206     |
| date (library function), configuring support for . . . . .    | 106      | diag_warning (pragma directive) . . . . .                          | 255     |
| DC32 (assembler directive) . . . . .                          | 124      | DIFUNCT (segment) . . . . .                                        | 76, 304 |
| --debug (compiler option) . . . . .                           | 203      | directives                                                         |         |
| debug information, including in object file . . . . .         | 203      | function for static overlay . . . . .                              | 135     |
| decimal point, implementation-defined behavior . . . . .      | 325      | pragma . . . . .                                                   | 41, 251 |
| declarations                                                  |          | directory, specifying as parameter . . . . .                       | 196     |
| empty . . . . .                                               | 151      | __disable_interrupt (intrinsic function) . . . . .                 | 271     |
| in for loops . . . . .                                        | 145      | --discard_unused_publics (compiler option) . . . . .               | 206     |
| Kernighan & Ritchie . . . . .                                 | 183      | disclaimer . . . . .                                               | 2       |
| of functions . . . . .                                        | 128      | DLIB . . . . .                                                     | 39, 285 |
| declarations and statements, mixing . . . . .                 | 145      | configurations . . . . .                                           | 96      |
| declarators, implementation-defined behavior in C89 . . . . . | 332      | configuring . . . . .                                              | 80, 207 |
| define_type_info (pragma directive) . . . . .                 | 319, 333 | documentation . . . . .                                            | 28      |
| delete operator (extended EC++) . . . . .                     | 159      | including debug support . . . . .                                  | 86      |
| delete (keyword) . . . . .                                    | 53       | reference information. <i>See</i> the online help system . . . . . | 283     |
| denormalized numbers. <i>See</i> subnormal numbers            |          | runtime environment . . . . .                                      | 79      |
| --dependencies (compiler option) . . . . .                    | 203      | --dlib (compiler option) . . . . .                                 | 207     |
| deque (STL header file) . . . . .                             | 287      | --dlib_config (compiler option) . . . . .                          | 207     |
| destructors and interrupts, using . . . . .                   | 160      | DLib_Defaults.h (library configuration file) . . . . .             | 92, 96  |
| DI (assembler instruction) . . . . .                          | 271      | __DLIB_FILE_DESCRIPTOR (configuration symbol) . . . . .            | 102     |
| diagnostic messages . . . . .                                 | 192      | document conventions . . . . .                                     | 29      |
| classifying as compilation errors . . . . .                   | 204      | documentation, overview of guides . . . . .                        | 27      |
| classifying as compilation remarks . . . . .                  | 205      | domain errors, implementation-defined behavior . . . . .           | 320     |

domain errors, implementation-defined behavior in C89 (CLIB) . . . . . 338  
 domain errors, implementation-defined behavior in C89 (DLIB) . . . . . 334  
 \_\_DOUBLE\_\_ (predefined symbol) . . . . . 279  
 double (data type) . . . . . 231  
   avoiding . . . . . 169  
   identifying size of (\_\_DOUBLE\_\_) . . . . . 279  
   in parameter passing . . . . . 130  
 do\_not\_instantiate (pragma directive) . . . . . 319, 333  
 dynamic initialization . . . . . 92, 117  
   and C++ . . . . . 76  
 dynamic memory . . . . . 53

## E

-e (compiler option) . . . . . 208  
 early\_initialization (pragma directive) . . . . . 319, 333  
 --ec++ (compiler option) . . . . . 208  
 edition, of this guide . . . . . 2  
 --eec++ (compiler option) . . . . . 209  
 EI (assembler instruction) . . . . . 271  
 EIWAIT (assembler instruction) . . . . . 271  
 Embedded C++ . . . . . 153  
   differences from C++ . . . . . 153  
   enabling . . . . . 208  
   function linkage . . . . . 128  
   language extensions . . . . . 153  
   overview . . . . . 153  
 Embedded C++ Technical Committee . . . . . 29  
 embedded systems, IAR special support for . . . . . 41  
 \_\_embedded\_cplusplus (predefined symbol) . . . . . 279  
 \_\_enable\_interrupt (intrinsic function) . . . . . 271  
 \_\_enable\_interrupt\_wait (intrinsic function) . . . . . 271  
 --enable\_multibytes (compiler option) . . . . . 209  
 entry label, program . . . . . 93  
 enumerations, implementation-defined behavior . . . . . 317  
 enumerations, implementation-defined behavior in C89 . . 331

enums  
   data representation . . . . . 228  
   forward declarations of . . . . . 150  
 environment  
   implementation-defined behavior . . . . . 312  
   implementation-defined behavior in C89 . . . . . 327  
   runtime (CLIB) . . . . . 113  
   runtime (DLIB) . . . . . 79  
 environment names, implementation-defined behavior . . 313  
 environment variables  
   C\_INCLUDE . . . . . 190  
   QCCCR16C . . . . . 190  
 environment (native),  
   implementation-defined behavior . . . . . 326  
 EQU (assembler directive) . . . . . 220  
 ERANGE . . . . . 321  
 ERANGE (C89) . . . . . 334  
 errno value at underflow,  
   implementation-defined behavior . . . . . 323  
 errno.h (library header file) . . . . . 286, 291  
 error messages . . . . . 193  
   classifying for compiler . . . . . 204  
 error return codes . . . . . 192  
 error (pragma directive) . . . . . 256  
 --error\_limit (compiler option) . . . . . 209  
 escape sequences, implementation-defined behavior . . . 313  
 exception flags, for floating-point values . . . . . 231  
 exception handling, missing from Embedded C++ . . . . 153  
 exception vectors . . . . . 76  
 EXCP (assembler instruction) . . . . . 57–58, 133, 249  
 \_Exit (library function) . . . . . 95  
 exit (library function) . . . . . 95  
   implementation-defined behavior . . . . . 323  
   implementation-defined behavior in C89 . . . . . 337, 339  
 \_exit (library function) . . . . . 95  
 \_\_exit (library function) . . . . . 95  
 exp (library routine) . . . . . 107  
 expf (library routine) . . . . . 108  
 expl (library routine) . . . . . 108  
 export keyword, missing from Extended EC++ . . . . . 161

|                                  |     |
|----------------------------------|-----|
| extended command line file       |     |
| for compiler                     | 210 |
| passing options                  | 189 |
| Extended Embedded C++            | 154 |
| enabling                         | 209 |
| extended keywords                | 239 |
| enabling (-e)                    | 208 |
| overview                         | 41  |
| summary                          | 242 |
| syntax                           | 48  |
| object attributes                | 242 |
| type attributes on data objects  | 240 |
| type attributes on data pointers | 241 |
| type attributes on functions     | 241 |
| extern "C" linkage               | 158 |

## F

|                                                                    |     |
|--------------------------------------------------------------------|-----|
| -f (compiler option)                                               | 210 |
| fatal error messages                                               | 194 |
| fdopen, in stdio.h                                                 | 289 |
| fegettrappable                                                     | 289 |
| fegetrapenable                                                     | 289 |
| FENV_ACCESS, implementation-defined behavior                       | 316 |
| fenv.h (library header file)                                       | 286 |
| additional C functionality                                         | 289 |
| fgetpos (library function), implementation-defined behavior        | 323 |
| fgetpos (library function), implementation-defined behavior in C89 | 336 |
| field width, library support for                                   | 116 |
| __FILE__ (predefined symbol)                                       | 279 |
| file buffering, implementation-defined behavior                    | 322 |
| file dependencies, tracking                                        | 203 |
| file paths, specifying for #include files                          | 211 |
| file position, implementation-defined behavior                     | 321 |
| file systems in C89                                                | 338 |
| file (zero-length), implementation-defined behavior                | 322 |

|                                                                 |               |
|-----------------------------------------------------------------|---------------|
| filename                                                        |               |
| extension for linker configuration file                         | 66            |
| of object file                                                  | 219           |
| search procedure for                                            | 190           |
| specifying as parameter                                         | 196           |
| filenames (legal), implementation-defined behavior              | 322           |
| fileno, in stdio.h                                              | 289           |
| files, implementation-defined behavior                          |               |
| handling of temporary                                           | 322           |
| multibyte characters in                                         | 322           |
| opening                                                         | 322           |
| float (data type)                                               | 231           |
| floating-point constants                                        |               |
| hexadecimal notation                                            | 145           |
| hints                                                           | 170           |
| floating-point environment, accessing or not                    | 265           |
| floating-point expressions                                      |               |
| contracting or not                                              | 266           |
| floating-point format                                           | 231           |
| hints                                                           | 170           |
| implementation-defined behavior                                 | 315           |
| implementation-defined behavior in C89                          | 330           |
| special cases                                                   | 232           |
| 32-bits                                                         | 232           |
| 64-bits                                                         | 232           |
| floating-point numbers, support for in printf formatters        | 116           |
| floating-point status flags                                     | 289           |
| float.h (library header file)                                   | 286, 291      |
| FLT_EVAL_METHOD, implementation-defined behavior                | 315, 320, 324 |
| FLT_ROUNDS, implementation-defined behavior                     | 315, 324      |
| fmod (library function), implementation-defined behavior in C89 | 335, 338      |
| for loops, declarations in                                      | 145           |
| formats                                                         |               |
| floating-point values                                           | 231           |
| standard IEEE (floating point)                                  | 231           |
| formatted_write (library function)                              | 115           |
| FP_CONTRACT, implementation-defined behavior                    | 316           |

fragmentation, of heap memory . . . . . 53  
 free (library function). *See also* heap . . . . . 53  
 fsetpos (library function), implementation-defined  
 behavior . . . . . 323  
 fstream (library header file) . . . . . 287  
 ftell (library function), implementation-defined behavior . 323  
 ftell (library function), implementation-defined behavior in  
 C89 . . . . . 336  
 Full DLIB (library configuration) . . . . . 97  
 \_\_func\_\_ (predefined symbol) . . . . . 152, 279  
 FUNCALL (assembler directive) . . . . . 135  
 \_\_FUNCTION\_\_ (predefined symbol) . . . . . 152, 280  
 function calls  
     calling convention . . . . . 128  
     normal code model . . . . . 135  
     short code model . . . . . 135  
     stack image after . . . . . 131  
 function declarations, Kernighan & Ritchie . . . . . 183  
 function directives for static overlay . . . . . 135  
 function inlining (compiler transformation) . . . . . 180  
     disabling (--no\_inline) . . . . . 214  
 function prototypes . . . . . 182  
     enforcing . . . . . 221  
 function return addresses . . . . . 132  
 function template parameter deduction (extended EC++) . 162  
 function type information, omitting in object output. . . . 218  
 FUNCTION (assembler directive) . . . . . 135  
 function (pragma directive) . . . . . 319, 333  
 functional (STL header file) . . . . . 287  
 functions . . . . . 55  
     calling in different code models . . . . . 135  
     C++ and special function types . . . . . 62  
     declaring . . . . . 128, 182  
     inlining . . . . . 145, 180, 182, 257  
     interrupt . . . . . 56, 58  
     intrinsic . . . . . 121, 182  
     monitor . . . . . 58  
     omitting type info . . . . . 218  
     parameters . . . . . 130  
     placing in memory . . . . . 173, 175, 222

recursive  
     avoiding . . . . . 182  
     storing data on stack . . . . . 52  
 reentrancy (DLIB) . . . . . 284  
 related extensions . . . . . 55  
 return values from . . . . . 131  
 special function types . . . . . 56  
 trap . . . . . 57  
     user . . . . . 58  
     verifying linked result . . . . . 76  
 function\_effects (pragma directive) . . . . . 319, 333

## G

getchar (library function) . . . . . 115  
 getenv (library function), configuring support for . . . . . 105  
 getw, in stdio.h . . . . . 289  
 getzone.c . . . . . 106  
 \_\_get\_interrupt\_state (intrinsic function) . . . . . 271  
 \_\_get\_processor\_register (intrinsic function) . . . . . 272  
 global arrays, accessing . . . . . 136  
 global base pointer register, considerations . . . . . 130  
 global variables  
     accessing . . . . . 136  
     initialization . . . . . 71  
 --guard\_calls (compiler option) . . . . . 210  
 guidelines, reading . . . . . 25

## H

Harbison, Samuel P. . . . . 28  
 hardware support in compiler . . . . . 79  
 hash\_map (STL header file) . . . . . 287  
 hash\_set (STL header file) . . . . . 287  
 \_\_has\_constructor, symbol used in library . . . . . 290  
 \_\_has\_destructor, symbol used in library . . . . . 290  
 hdrstop (pragma directive) . . . . . 319, 333  
 header files  
     C . . . . . 285

|                                                    |         |                                        |     |
|----------------------------------------------------|---------|----------------------------------------|-----|
| C++                                                | 286–287 | __iar_cos_accuratel (library routine)  | 109 |
| library                                            | 283     | __iar_cos_accuratel (library function) | 284 |
| special function registers                         | 185     | __iar_cos_small (library routine)      | 107 |
| STL                                                | 287     | __iar_cos_smallf (library routine)     | 108 |
| DLib_Defaults.h                                    | 92, 96  | __iar_cos_smallll (library routine)    | 108 |
| including stdbool.h for bool                       | 228     | __iar_exp_small (library routine)      | 107 |
| including stddef.h for wchar_t                     | 229     | __iar_exp_smallf (library routine)     | 108 |
| header names, implementation-defined behavior      | 318     | __iar_exp_smallll (library routine)    | 108 |
| --header_context (compiler option)                 | 210     | __iar_log_small (library routine)      | 107 |
| heap                                               |         | __iar_log_smallf (library routine)     | 108 |
| DLIB support for multiple                          | 109     | __iar_log_smallll (library routine)    | 108 |
| dynamic memory                                     | 53      | __iar_log10_small (library routine)    | 107 |
| segments for                                       | 73      | __iar_log10_smallf (library routine)   | 108 |
| storing data                                       | 44      | __iar_log10_smallll (library routine)  | 108 |
| VLA allocated on                                   | 224     | __iar_Powf (library routine)           | 109 |
| heap segments                                      |         | __iar_Powl (library routine)           | 109 |
| CLIB                                               | 74      | __iar_Pow_accurate (library routine)   | 108 |
| DATA16_HEAP (segment)                              | 297     | __iar_pow_accurate (library routine)   | 108 |
| DATA32_HEAP (segment)                              | 302     | __iar_Pow_accuratef (library routine)  | 109 |
| DLIB                                               | 73      | __iar_pow_accuratef (library routine)  | 109 |
| placing                                            | 74      | __iar_pow_accuratef (library function) | 284 |
| heap size                                          |         | __iar_Pow_accuratel (library routine)  | 109 |
| and standard I/O                                   | 74      | __iar_pow_accuratel (library routine)  | 109 |
| changing default                                   | 74      | __iar_pow_accuratel (library function) | 284 |
| heap (zero-sized), implementation-defined behavior | 323     | __iar_pow_small (library routine)      | 107 |
| hints                                              |         | __iar_pow_smallf (library routine)     | 108 |
| for good code generation                           | 181     | __iar_pow_smallll (library routine)    | 108 |
| implementation-defined behavior                    | 317     | __iar_Sin (library routine)            | 107 |
| using efficient data types                         | 169     | __iar_Sinfl (library routine)          | 109 |
|                                                    |         | __iar_Sin_accurate (library routine)   | 108 |
|                                                    |         | __iar_sin_accurate (library routine)   | 108 |
|                                                    |         | __iar_Sin_accuratef (library routine)  | 109 |
|                                                    |         | __iar_sin_accuratef (library routine)  | 109 |
|                                                    |         | __iar_sin_accuratef (library function) | 284 |
|                                                    |         | __iar_Sin_accuratel (library routine)  | 109 |
|                                                    |         | __iar_sin_accuratel (library routine)  | 109 |
|                                                    |         | __iar_sin_accuratel (library function) | 284 |
|                                                    |         | __iar_Sin_small (library routine)      | 107 |
| -I (compiler option)                               | 211     |                                        |     |
| IAR Command Line Build Utility                     | 91      |                                        |     |
| IAR Systems Technical Support                      | 194     |                                        |     |
| iarbuild.exe (utility)                             | 91      |                                        |     |
| __iar_cos_accurate (library routine)               | 108     |                                        |     |
| __iar_cos_accuratef (library routine)              | 109     |                                        |     |
| __iar_cos_accuratef (library function)             | 284     |                                        |     |

- \_\_jar\_sin\_small (library routine) . . . . . 107
- \_\_jar\_Sin\_smallf (library routine) . . . . . 108
- \_\_jar\_sin\_smallf (library routine) . . . . . 108
- \_\_jar\_Sin\_smalll (library routine) . . . . . 108
- \_\_jar\_sin\_smalll (library routine) . . . . . 108
- \_\_IAR\_SYSTEMS\_ICC\_\_ (predefined symbol) . . . . . 280
- \_\_jar\_tan\_accurate (library routine) . . . . . 108
- \_\_jar\_tan\_accuratef (library routine) . . . . . 109
- \_\_jar\_tan\_accuratef (library function) . . . . . 284
- \_\_jar\_tan\_accuratel (library routine) . . . . . 109
- \_\_jar\_tan\_accuratel (library function) . . . . . 284
- \_\_jar\_tan\_small (library routine) . . . . . 107
- \_\_jar\_tan\_smallf (library routine) . . . . . 108
- \_\_jar\_tan\_smalll (library routine) . . . . . 108
- iccbutl.h (library header file) . . . . . 291
- \_\_ICCCR16C\_\_ (predefined symbol) . . . . . 280
- icons, in this guide . . . . . 30
- IDE
  - building a library from . . . . . 91
  - building applications from, an overview . . . . . 36
- identifiers, implementation-defined behavior . . . . . 313
- identifiers, implementation-defined behavior in C89 . . . . 328
- IDINIT (segment) . . . . . 304
- IEEE format, floating-point values . . . . . 231
- implementation-defined behavior
  - C89 . . . . . 327
  - Standard C . . . . . 311
- important typedef (pragma directive) . . . . . 319, 333
- include files
  - including before source files . . . . . 219
  - specifying . . . . . 190
- include\_alias (pragma directive) . . . . . 256
- \_\_INDEXED\_ENABLED\_\_ (predefined symbol) . . . . . 280
- INDEX\_BASE (segment) . . . . . 304
- index20 (memory type) . . . . . 47
- INDEX20\_C (segment) . . . . . 308
- INDEX20\_I (segment) . . . . . 308
- INDEX20\_ID (segment) . . . . . 308
- INDEX20\_Z (segment) . . . . . 309
- index4 (memory type) . . . . . 47
- INDEX4\_I (segment) . . . . . 304
- INDEX4\_ID (segment) . . . . . 305
- INDEX4\_N (segment) . . . . . 305, 309
- INDEX4\_Z (segment) . . . . . 305
- index5 (memory type) . . . . . 47
- INDEX5\_I (segment) . . . . . 306
- INDEX5\_ID (segment) . . . . . 306
- INDEX5\_N (segment) . . . . . 307
- INDEX5\_Z (segment) . . . . . 307
- infinity . . . . . 232
- infinity (style for printing), implementation-defined
  - behavior . . . . . 323
  - inheritance, in Embedded C++ . . . . . 153
- initialization
  - dynamic . . . . . 92, 117
  - single-value . . . . . 151
- initialized data segments . . . . . 71
- initializers, static . . . . . 150
- inline assembler . . . . . 123
  - avoiding . . . . . 182
  - See also* assembler language interface
- inline functions . . . . . 145
  - in compiler . . . . . 180
- inline (pragma directive) . . . . . 257
- inlining functions, implementation-defined behavior . . . . 317
- installation directory . . . . . 29
- instantiate (pragma directive) . . . . . 319, 333
- int (data type) signed and unsigned . . . . . 228
- integer types . . . . . 228
  - casting . . . . . 233
  - implementation-defined behavior . . . . . 315
- intptr\_t . . . . . 234
- ptrdiff\_t . . . . . 234
- size\_t . . . . . 234
- uintptr\_t . . . . . 234
- integers, implementation-defined behavior in C89 . . . . . 329
- integral promotion . . . . . 183
- internal error . . . . . 194

|                                                         |         |
|---------------------------------------------------------|---------|
| __interrupt (extended keyword)                          | 57, 245 |
| using in pragma directives                              | 267     |
| interrupt functions                                     | 56      |
| placement in memory                                     | 76      |
| interrupt handler. <i>See</i> interrupt service routine |         |
| interrupt service routine                               | 56      |
| interrupt state, restoring                              | 274     |
| interrupt vector                                        | 56      |
| specifying with pragma directive                        | 267     |
| interrupt vector table                                  | 56–57   |
| IDINIT segment                                          | 304     |
| in linker configuration file                            | 76      |
| INDEX_BASE segment                                      | 304     |
| INTVEC segment                                          | 309     |
| ISTACK segment                                          | 309     |
| start address for                                       | 56      |
| ZINIT segment                                           | 310     |
| interrupts                                              |         |
| disabling                                               | 247     |
| during function execution                               | 58      |
| processor state                                         | 52      |
| using with EC++ destructors                             | 160     |
| intptr_t (integer type)                                 | 234     |
| __intrinsic (extended keyword)                          | 246     |
| intrinsic functions                                     | 182     |
| overview                                                | 121     |
| summary                                                 | 269     |
| intrinsics.h (header file)                              | 269     |
| inttypes.h (library header file)                        | 286     |
| INTVEC (segment)                                        | 76, 309 |
| intwri.c (library source code)                          | 116     |
| invocation syntax                                       | 189     |
| ioomanip (library header file)                          | 287     |
| ios (library header file)                               | 287     |
| iosfwd (library header file)                            | 287     |
| iostream (library header file)                          | 287     |
| iso646.h (library header file)                          | 286     |
| ISTACK (segment)                                        | 309     |
| istream (library header file)                           | 287     |

|                              |     |
|------------------------------|-----|
| italic style, in this guide  | 30  |
| iterator (STL header file)   | 287 |
| __ix20 (extended keyword)    | 246 |
| __ix4 (extended keyword)     | 246 |
| I/O register. <i>See</i> SFR |     |
| I/O, character-based         | 115 |

## J

|                              |     |
|------------------------------|-----|
| Josuttis, Nicolai M.         | 28  |
| JUSR (assembler instruction) | 249 |

## K

|                                           |          |
|-------------------------------------------|----------|
| keep_definition (pragma directive)        | 319, 333 |
| Kernighan & Ritchie function declarations | 183      |
| disallowing                               | 221      |
| Kernighan, Brian W.                       | 28       |
| keywords                                  | 239      |
| extended, overview of                     | 41       |

## L

|                                 |     |
|---------------------------------|-----|
| -l (compiler option)            | 211 |
| for creating skeleton code      | 126 |
| labels                          | 151 |
| assembler, making public        | 220 |
| __program_start                 | 93  |
| Labrosse, Jean J.               | 29  |
| Lajoie, Josée                   | 29  |
| language extensions             |     |
| Embedded C++                    | 153 |
| enabling using pragma           | 257 |
| enabling (-e)                   | 208 |
| language overview               | 35  |
| language (pragma directive)     | 257 |
| _large_write (library function) | 116 |
| libraries                       |     |
| definition of                   | 36  |



- standard template library . . . . . 287
- using a prebuilt . . . . . 81
- using a prebuilt (CLIB) . . . . . 113
- library configuration files
  - DLIB . . . . . 96
  - DLib\_Defaults.h . . . . . 92, 96
  - modifying . . . . . 92
  - specifying . . . . . 207
- library documentation . . . . . 283
- library features, missing from Embedded C++ . . . . . 154
- library functions . . . . . 283
  - summary, CLIB . . . . . 291
  - summary, DLIB . . . . . 285
- library header files . . . . . 283
- library modules
  - creating . . . . . 212
  - overriding . . . . . 90
- library object files . . . . . 284
- library options, setting . . . . . 40
- library project template . . . . . 39
  - using . . . . . 91
- library\_default\_requirements (pragma directive) . . . 319, 333
- library\_module (compiler option) . . . . . 212
- library\_provides (pragma directive) . . . . . 319, 334
- library\_requirement\_override (pragma directive) . . . 319, 334
- lightbulb icon, in this guide . . . . . 30
- limits.h (library header file) . . . . . 286, 291
- \_\_LINE\_\_ (predefined symbol) . . . . . 280
- link register, considerations . . . . . 130
- linkage, C and C++ . . . . . 128
- linker configuration file . . . . . 66
  - customizing . . . . . 66
  - using the -P command . . . . . 68
  - using the -Z command . . . . . 68
- linker map file . . . . . 77
- linker output files . . . . . 37
- linker segment. *See* segment
- linking
  - from the command line . . . . . 37
  - required input . . . . . 37
- Lippman, Stanley B. . . . . 29
- list (STL header file) . . . . . 287
- listing, generating . . . . . 211
- literals, compound . . . . . 145
- literature, recommended . . . . . 28
- \_\_LITTLE\_ENDIAN\_\_ (predefined symbol) . . . . . 280
- local variables, *See* auto variables
- locale
  - adding support for in library . . . . . 104
  - changing at runtime . . . . . 104
  - implementation-defined behavior . . . . . 314, 325
  - removing support for . . . . . 104
  - support for . . . . . 103
- locale.h (library header file) . . . . . 286
- located data segments . . . . . 75
- located data, declaring extern . . . . . 175
- location (pragma directive) . . . . . 174, 258
- LOCFRAME (assembler directive) . . . . . 135
- log (library routine) . . . . . 107
- logf (library routine) . . . . . 108
- logl (library routine) . . . . . 108
- log10 (library routine) . . . . . 107
- log10f (library routine) . . . . . 108
- log10l (library routine) . . . . . 108
- long double (data type) . . . . . 231
- long float (data type), synonym for double . . . . . 150
- long long (data type)
  - restrictions . . . . . 228
- long long (data type) signed and unsigned . . . . . 228
- long (data type) signed and unsigned . . . . . 228
- longjmp, restrictions for using . . . . . 285
- loop unrolling (compiler transformation) . . . . . 180
  - disabling . . . . . 216
- loop-invariant expressions . . . . . 180
- \_\_low\_level\_init . . . . . 93
  - customizing . . . . . 96
- low-level processor operations . . . . . 146, 269
  - accessing . . . . . 121

\_\_lseek (library function) . . . . . 102

## M

MACQW (assembler instruction) . . . . . 272

### macros

    embedded in #pragma optimize . . . . . 260

    ERANGE (in errno.h) . . . . . 321, 334

    inclusion of assert . . . . . 282

    NULL, implementation-defined behavior . . . . . 321

        in C89 for CLIB . . . . . 337

        in C89 for DLIB . . . . . 334

    substituted in #pragma directives . . . . . 146

    variadic . . . . . 145

--macro\_positions\_in\_diagnostics (compiler option) . . . . . 212

MACSW (assembler instruction) . . . . . 273

MACUW (assembler instruction) . . . . . 273

\_\_mac\_q15 (intrinsic function) . . . . . 272

\_\_mac\_signed (intrinsic function) . . . . . 273

\_\_mac\_unsigned (intrinsic function) . . . . . 273

### main (function)

    definition (C89) . . . . . 327

    implementation-defined behavior . . . . . 312

### malloc (library function)

*See also* heap . . . . . 53

    implementation-defined behavior in C89 . . . . . 336, 339

Mann, Bernhard . . . . . 29

map (STL header file) . . . . . 287

map, linker . . . . . 77

### math functions rounding mode,

    implementation-defined behavior . . . . . 324

    math functions (library functions) . . . . . 107

    math.h (library header file) . . . . . 286, 291

    MB\_LEN\_MAX, implementation-defined behavior . . . . . 324

    \_\_medium\_write (library function) . . . . . 116

    \_\_memcpy\_generic (intrinsic function) . . . . . 273

### memory

    accessing . . . . . 38, 46, 136

        using data16 method . . . . . 136

        using data20 method . . . . . 137

        using data24 method . . . . . 137

        using data32 method . . . . . 138

    allocating in C++ . . . . . 53

    dynamic . . . . . 53

    heap . . . . . 53

    non-initialized . . . . . 185

    RAM, saving . . . . . 182

    releasing in C++ . . . . . 53

    stack . . . . . 52

        saving . . . . . 182

    used by global or static variables . . . . . 43

    memory consumption, reducing . . . . . 115

    memory layout, CR16C . . . . . 43

    memory management, type-safe . . . . . 153

    memory map, customizing the linker configuration file for . . . . . 67

### memory placement

    using pragma directive . . . . . 49

    using type definitions . . . . . 49, 241

    memory segment. *See* segment

    memory types . . . . . 46

        C++ . . . . . 51

        placing variables in . . . . . 51

        pointers . . . . . 49

        specifying . . . . . 47

        structures . . . . . 50

        summary . . . . . 48

    memory (pragma directive) . . . . . 319, 334

    memory (STL header file) . . . . . 287

### \_\_memory\_of

    operator . . . . . 157

    symbol used in library . . . . . 290

\_\_memset\_generic (intrinsic function) . . . . . 273

message (pragma directive) . . . . . 259

### messages

    disabling . . . . . 223

    forcing . . . . . 259

    Meyers, Scott . . . . . 29

    --mfc (compiler option) . . . . . 213

migration, from earlier IAR compilers . . . . . 28

MISRA C, documentation . . . . . 28

--misrac (compiler option) . . . . . 198

--misrac\_verbose (compiler option) . . . . . 199

--misrac1998 (compiler option) . . . . . 199

--misrac2004 (compiler option) . . . . . 199

mode changing, implementation-defined behavior . . . . . 322

module consistency . . . . . 110

    rtmodel . . . . . 263

module map, in linker map file . . . . . 77

module name, specifying (--module\_name) . . . . . 213

module summary, in linker map file . . . . . 77

--module\_name (compiler option) . . . . . 213

module\_name (pragma directive) . . . . . 319, 334

\_\_monitor (extended keyword) . . . . . 247

monitor functions . . . . . 58, 247

multibyte character support . . . . . 209

multibyte characters, implementation-defined behavior . . . . . 313, 325

multiple inheritance

    missing from Embedded C++ . . . . . 153

    missing from STL . . . . . 154

multi-file compilation . . . . . 177

mutable attribute, in Extended EC++ . . . . . 154, 165

## N

names block (call frame information) . . . . . 140

namespace support

    in Extended EC++ . . . . . 154, 165

    missing from Embedded C++ . . . . . 154

naming conventions . . . . . 30

NaN

    implementation of . . . . . 232

    implementation-defined behavior . . . . . 323

native environment,

    implementation-defined behavior . . . . . 326

NDEBUG (preprocessor symbol) . . . . . 282

new operator (extended EC++) . . . . . 159

new (keyword) . . . . . 53

new (library header file) . . . . . 287

\_\_noadjust (extended keyword) . . . . . 247

non-initialized variables, hints for . . . . . 186

non-scalar parameters, avoiding . . . . . 182

NOP (assembler instruction) . . . . . 274

\_\_noreturn (extended keyword) . . . . . 248

Normal DLIB (library configuration) . . . . . 97

Normal register mode . . . . . 55

Not a number (NaN) . . . . . 232

--no\_code\_motion (compiler option) . . . . . 213

--no\_cse (compiler option) . . . . . 214

\_\_no\_init (extended keyword) . . . . . 186, 247

--no\_inline (compiler option) . . . . . 214

\_\_no\_operation (intrinsic function) . . . . . 274

--no\_path\_in\_file\_macros (compiler option) . . . . . 214

no\_pch (pragma directive) . . . . . 319, 334

--no\_size\_constraints (compiler option) . . . . . 215

--no\_static\_destruction (compiler option) . . . . . 215

--no\_system\_include (compiler option) . . . . . 215

--no\_tbaa (compiler option) . . . . . 216

--no\_typedefs\_in\_diagnostics (compiler option) . . . . . 216

--no\_unroll (compiler option) . . . . . 216

--no\_warnings (compiler option) . . . . . 217

--no\_wrap\_diagnostics (compiler option) . . . . . 217

NULL

    implementation-defined behavior . . . . . 321

    implementation-defined behavior in C89 (CLIB) . . . . . 337

    implementation-defined behavior in C89 (DLIB) . . . . . 334

    in library header file (CLIB) . . . . . 291

    pointer constant, relaxation to Standard C . . . . . 150

numeric conversion functions,

    implementation-defined behavior . . . . . 326

    numeric (STL header file) . . . . . 287

## O

-O (compiler option) . . . . . 217

-o (compiler option) . . . . . 218

|                                                |          |
|------------------------------------------------|----------|
| object attributes                              | 242      |
| object filename, specifying (-o)               | 219      |
| object module name, specifying (--module_name) | 213      |
| object_attribute (pragma directive)            | 186, 259 |
| offsetof                                       | 291      |
| --omit_types (compiler option)                 | 218      |
| once (pragma directive)                        | 319, 334 |
| --only_stdout (compiler option)                | 218      |
| __open (library function)                      | 102      |
| operators                                      |          |
| <i>See also</i> @ (operator)                   |          |
| for cast                                       |          |
| in Extended EC++                               | 154      |
| missing from Embedded C++                      | 154      |
| for segment control                            | 149      |
| in inline assembler                            | 123      |
| new and delete                                 | 159      |
| precision for 32-bit float                     | 232      |
| precision for 64-bit float                     | 232      |
| sizeof, implementation-defined behavior        | 325      |
| variants for cast                              | 165      |
| _Pragma (preprocessor)                         | 145      |
| __ALIGNOF__, for alignment control             | 148      |
| __memory_of.                                   | 157      |
| ?, language extensions for                     | 166      |
| optimization                                   |          |
| code motion, disabling                         | 213      |
| common sub-expression elimination, disabling   | 214      |
| configuration                                  | 39       |
| disabling                                      | 179      |
| function inlining, disabling (--no_inline)     | 214      |
| hints                                          | 181      |
| loop unrolling, disabling                      | 216      |
| specifying (-O)                                | 217      |
| techniques                                     | 179      |
| type-based alias analysis, disabling (--tbaa)  | 216      |
| using inline assembler code                    | 124      |
| using pragma directive                         | 260      |
| optimization levels                            | 178      |

|                                                |     |
|------------------------------------------------|-----|
| optimize (pragma directive)                    | 260 |
| option parameters                              | 195 |
| options, compiler. <i>See</i> compiler options |     |
| Oram, Andy                                     | 28  |
| ostream (library header file)                  | 287 |
| output                                         |     |
| from preprocessor                              | 220 |
| specifying for linker                          | 37  |
| supporting non-standard                        | 116 |
| --output (compiler option)                     | 218 |
| overhead, reducing                             | 180 |

## P

|                                                                      |          |
|----------------------------------------------------------------------|----------|
| pack (pragma directive)                                              | 235, 261 |
| packed structure types                                               | 235      |
| parameters                                                           |          |
| function                                                             | 130      |
| hidden                                                               | 130      |
| non-scalar, avoiding                                                 | 182      |
| register                                                             | 130      |
| rules for specifying a file or directory                             | 196      |
| specifying                                                           | 197      |
| stack                                                                | 130–131  |
| typographic convention                                               | 30       |
| part number, of this guide                                           | 2        |
| permanent registers                                                  | 129      |
| perorr (library function),<br>implementation-defined behavior in C89 | 336, 339 |
| placement                                                            |          |
| code and data                                                        | 293      |
| in named segments                                                    | 175      |
| plain char, implementation-defined behavior                          | 314      |
| pointer types                                                        | 233      |
| differences between                                                  | 50       |
| mixing                                                               | 150      |
| pointers                                                             |          |
| casting                                                              | 233      |
| code                                                                 | 233      |

- data . . . . . 233
- implementation-defined behavior. . . . . 316
- implementation-defined behavior in C89. . . . . 331
- polymorphism, in Embedded C++ . . . . . 153
- porting, code containing pragma directives. . . . . 252
- pow (library routine). . . . . 107–108
  - alternative implementation of. . . . . 284
- powf (library routine) . . . . . 108–109
- powl (library routine) . . . . . 108–109
- pragma directives . . . . . 41
  - summary . . . . . 251
  - basic\_template\_matching, using . . . . . 163
  - for absolute located data . . . . . 174
  - list of all recognized. . . . . 319
  - list of all recognized (C89). . . . . 333
  - pack . . . . . 235, 261
  - type\_attribute, using. . . . . 49
- \_Pragma (preprocessor operator) . . . . . 145
- precision arguments, library support for . . . . . 116
- predefined symbols
  - overview . . . . . 41
  - summary . . . . . 278
- predef\_macro (compiler option). . . . . 219
- preinclude (compiler option) . . . . . 219
- preprocess (compiler option) . . . . . 220
- preprocessor
  - operator (\_Pragma) . . . . . 145
  - output. . . . . 220
  - overview of . . . . . 277
- preprocessor directives
  - comments at the end of . . . . . 151
  - implementation-defined behavior. . . . . 318
  - implementation-defined behavior in C89. . . . . 332
  - #pragma . . . . . 251
- preprocessor extensions
  - \_\_VA\_ARGS\_\_ . . . . . 145
  - #warning message . . . . . 282
- preprocessor symbols . . . . . 278
  - defining . . . . . 202

- preserved registers . . . . . 129
- \_\_PRETTY\_FUNCTION\_\_ (predefined symbol). . . . . 280
- primitives, for special functions . . . . . 56
- print formatter, selecting. . . . . 85
- printf (library function). . . . . 84, 115
  - choosing formatter . . . . . 84
  - configuration symbols . . . . . 100
  - customizing . . . . . 116
  - implementation-defined behavior. . . . . 323
  - implementation-defined behavior in C89 . . . . . 336, 339
  - selecting. . . . . 116
- \_\_printf\_args (pragma directive). . . . . 262
- printing characters, implementation-defined behavior . . . 325
- processor operations
  - accessing . . . . . 121
  - low-level . . . . . 146, 269
- processor, getting the value of (\_\_get\_processor\_register) 272
- program entry label. . . . . 93
- program termination, implementation-defined behavior . . 312
- programming hints . . . . . 181
- \_\_program\_start (label). . . . . 93
- program, *see also* application
- projects
  - basic settings for . . . . . 37
  - setting up for a library . . . . . 91
- prototypes, enforcing . . . . . 221
- ptrdiff\_t (integer type). . . . . 234, 291
- PUBLIC (assembler directive) . . . . . 220
- publication date, of this guide. . . . . 2
- public\_equ (compiler option) . . . . . 220
- public\_equ (pragma directive) . . . . . 319, 334
- putchar (library function) . . . . . 115
- putenv (library function), absent from DLIB . . . . . 105
- putw, in stdio.h . . . . . 289

## Q

- QCCCR16C (environment variable). . . . . 190

|                                        |     |
|----------------------------------------|-----|
| qualifiers                             |     |
| const and volatile                     | 236 |
| implementation-defined behavior        | 318 |
| implementation-defined behavior in C89 | 332 |
| queue (STL header file)                | 288 |

## R

|                                                   |          |
|---------------------------------------------------|----------|
| -r (compiler option)                              | 203      |
| raise (library function), configuring support for | 106      |
| __raise_exception (intrinsic function)            | 274      |
| raise.c                                           | 106      |
| RAM                                               |          |
| non-zero initialized variables                    | 71       |
| saving memory                                     | 182      |
| range errors, in linker                           | 76       |
| __raw (extended keyword)                          | 248      |
| __read (library function)                         | 102      |
| customizing                                       | 98       |
| read formatter, selecting                         | 86, 117  |
| reading guidelines                                | 25       |
| reading, recommended                              | 28       |
| realloc (library function)                        | 53       |
| implementation-defined behavior in C89            | 336, 339 |
| <i>See also</i> heap                              |          |
| recursive functions                               |          |
| avoiding                                          | 182      |
| storing data on stack                             | 52       |
| reentrancy (DLIB)                                 | 284      |
| reference information, typographic convention     | 30       |
| register keyword, implementation-defined behavior | 317      |
| register modes                                    | 55       |
| register parameters                               | 130      |
| registered trademarks                             | 2        |
| registers                                         |          |
| assigning to parameters                           | 131      |
| callee-save, stored on stack                      | 52       |
| for function returns                              | 132      |
| implementation-defined behavior in C89            | 331      |

|                                                 |               |
|-------------------------------------------------|---------------|
| in assembler-level routines                     | 128           |
| preserved                                       | 129           |
| processor,                                      |               |
| getting the value of (__get_processor_register) | 272           |
| scratch                                         | 129           |
| reinterpret_cast (cast operator)                | 154           |
| --relaxed_fp (compiler option)                  | 220           |
| remark (diagnostic message)                     | 193           |
| classifying for compiler                        | 205           |
| enabling in compiler                            | 221           |
| --remarks (compiler option)                     | 221           |
| remove (library function)                       | 102           |
| implementation-defined behavior                 | 322           |
| implementation-defined behavior in C89 (CLIB)   | 339           |
| implementation-defined behavior in C89 (DLIB)   | 336           |
| remquo, magnitude of                            | 321           |
| rename (library function)                       | 102           |
| implementation-defined behavior                 | 322           |
| implementation-defined behavior in C89 (CLIB)   | 339           |
| implementation-defined behavior in C89 (DLIB)   | 336           |
| __ReportAssert (library function)               | 109           |
| required (pragma directive)                     | 262           |
| --require_prototypes (compiler option)          | 221           |
| return addresses                                | 132           |
| return values, from functions                   | 131           |
| RETX (assembler instruction)                    | 249           |
| Ritchie, Dennis M.                              | 28            |
| __root (extended keyword)                       | 248           |
| routines, time-critical                         | 121, 146, 269 |
| rtmodel (assembler directive)                   | 111           |
| rtmodel (pragma directive)                      | 263           |
| rti support, missing from STL                   | 154           |
| __rt_version (runtime model attribute)          | 111           |
| runtime environment                             |               |
| CLIB                                            | 113           |
| DLIB                                            | 79            |
| setting options for                             | 40            |
| setting up (DLIB)                               | 80            |
| runtime libraries (CLIB)                        |               |
| introduction                                    | 283           |

|                                                     |     |
|-----------------------------------------------------|-----|
| filename syntax                                     | 114 |
| using prebuilt                                      | 113 |
| runtime libraries (DLIB)                            |     |
| introduction                                        | 283 |
| customizing system startup code                     | 95  |
| customizing without rebuilding                      | 83  |
| filename syntax                                     | 83  |
| overriding modules in                               | 90  |
| using prebuilt                                      | 81  |
| runtime library                                     |     |
| setting up from command line                        | 40  |
| setting up from IDE                                 | 40  |
| runtime model attributes                            | 110 |
| runtime model definitions                           | 263 |
| runtime type information, missing from Embedded C++ | 153 |

## S

|                                                   |          |
|---------------------------------------------------|----------|
| scanf (library function)                          |          |
| choosing formatter (CLIB)                         | 117      |
| choosing formatter (DLIB)                         | 85       |
| configuration symbols                             | 100      |
| implementation-defined behavior                   | 323      |
| implementation-defined behavior in C89            | 339      |
| implementation-defined behavior in C89 (CLIB)     | 339      |
| implementation-defined behavior in C89 (DLIB)     | 336      |
| __scanf_args (pragma directive)                   | 263      |
| scratch registers                                 | 129      |
| section (pragma directive)                        | 264      |
| segment group name                                | 69       |
| segment map, in linker map file                   | 77       |
| segment memory types, in XLINK                    | 66       |
| --segment (compiler option)                       | 222      |
| segment (pragma directive)                        | 264      |
| segments                                          | 293      |
| CODE                                              | 76       |
| code                                              | 75       |
| data                                              | 69       |
| DATA16_HEAP                                       | 74       |
| DATA32_HEAP                                       | 74       |
| declaring (#pragma segment)                       | 264      |
| definition of                                     | 65       |
| initialized data                                  | 71       |
| introduction                                      | 65       |
| located data                                      | 75       |
| naming                                            | 70       |
| packing in memory                                 | 68       |
| placing in sequence                               | 68       |
| specifying (--segment)                            | 222      |
| static memory                                     | 69       |
| summary                                           | 293      |
| too long for address range                        | 76       |
| too long, in linker                               | 76       |
| __segment_begin (extended operator)               | 149      |
| __segment_end (extended operator)                 | 149      |
| __segment_size (extended operator)                | 149      |
| semaphores                                        |          |
| C example                                         | 58       |
| C++ example                                       | 60       |
| operations on                                     | 247      |
| set (STL header file)                             | 288      |
| setjmp.h (library header file)                    | 286, 291 |
| setlocale (library function)                      | 104      |
| settings, basic for project configuration         | 37       |
| __set_interrupt_state (intrinsic function)        | 274      |
| __set_processor_register (intrinsic function)     | 274      |
| __set_processor_register_bit (intrinsic function) | 274      |
| __set_PSR_I_bit (intrinsic function)              | 275      |
| severity level, of diagnostic messages            | 193      |
| specifying                                        | 194      |
| SFR                                               |          |
| accessing special function registers              | 185      |
| declaring extern special function registers       | 175      |
| shared object                                     | 192      |
| Short register mode                               | 55       |
| short (data type)                                 | 228      |
| signal (library function)                         |          |
| configuring support for                           | 106      |

|                                                                    |          |                                                              |          |
|--------------------------------------------------------------------|----------|--------------------------------------------------------------|----------|
| implementation-defined behavior. . . . .                           | 321      | stack . . . . .                                              | 52, 72   |
| implementation-defined behavior in C89. . . . .                    | 335      | advantages and problems using . . . . .                      | 52       |
| signals, implementation-defined behavior. . . . .                  | 312      | changing default size of . . . . .                           | 72       |
| at system startup . . . . .                                        | 313      | cleaning after function return . . . . .                     | 132      |
| signal.c . . . . .                                                 | 106      | contents of . . . . .                                        | 52       |
| signal.h (library header file) . . . . .                           | 286      | layout . . . . .                                             | 131      |
| signed char (data type) . . . . .                                  | 228–229  | saving space . . . . .                                       | 182      |
| specifying . . . . .                                               | 201      | size. . . . .                                                | 73       |
| signed int (data type). . . . .                                    | 228      | stack parameters . . . . .                                   | 130–131  |
| signed long long (data type) . . . . .                             | 228      | stack pointer . . . . .                                      | 52       |
| signed long (data type) . . . . .                                  | 228      | stack pointer register, considerations. . . . .              | 130      |
| signed short (data type). . . . .                                  | 228      | stack segment                                                |          |
| signed values, avoiding . . . . .                                  | 169      | CSTACK . . . . .                                             | 296      |
| --silent (compiler option) . . . . .                               | 223      | placing in memory . . . . .                                  | 73       |
| silent operation                                                   |          | stack (STL header file) . . . . .                            | 288      |
| specifying in compiler . . . . .                                   | 223      | Standard C                                                   |          |
| sin (library function) . . . . .                                   | 284      | implementation-defined behavior. . . . .                     | 311      |
| sin (library routine) . . . . .                                    | 107–108  | library compliance with . . . . .                            | 39, 283  |
| sinf (library routine) . . . . .                                   | 108–109  | specifying strict usage . . . . .                            | 223      |
| sinl (library routine) . . . . .                                   | 108–109  | standard error                                               |          |
| 64-bits (floating-point format) . . . . .                          | 232      | redirecting in compiler. . . . .                             | 218      |
| size_t (integer type) . . . . .                                    | 234, 291 | standard input . . . . .                                     | 98       |
| skeleton code, creating for assembler language interface . . . . . | 125      | standard output . . . . .                                    | 98       |
| skeleton.s45 (assembler source output). . . . .                    | 126      | specifying in compiler . . . . .                             | 218      |
| slist (STL header file) . . . . .                                  | 288      | standard template library (STL)                              |          |
| _small_write (library function) . . . . .                          | 116      | in C++ . . . . .                                             | 287      |
| source files, list all referred. . . . .                           | 210      | in Extended EC++ . . . . .                                   | 154, 161 |
| space characters, implementation-defined behavior . . . . .        | 321      | missing from Embedded C++ . . . . .                          | 154      |
| special function registers (SFR) . . . . .                         | 185      | startup code                                                 |          |
| special function types . . . . .                                   | 56       | placement of . . . . .                                       | 75       |
| overview . . . . .                                                 | 41       | <i>See also</i> CSTART                                       |          |
| sprintf (library function) . . . . .                               | 84, 115  | startup system. <i>See</i> system startup                    |          |
| choosing formatter. . . . .                                        | 84       | statements, implementation-defined behavior in C89 . . . . . | 332      |
| customizing . . . . .                                              | 116      | static data, in configuration file. . . . .                  | 72       |
| sscanf (library function)                                          |          | static memory segments . . . . .                             | 69       |
| choosing formatter (CLIB). . . . .                                 | 117      | static overlay. . . . .                                      | 134      |
| choosing formatter (DLIB) . . . . .                                | 85       | static variables . . . . .                                   | 43       |
| sstream (library header file) . . . . .                            | 287      | initialization. . . . .                                      | 71       |
|                                                                    |          | taking the address of . . . . .                              | 182      |



- static\_assert() . . . . . 148
- static\_cast (cast operator) . . . . . 154
- status flags for floating-point . . . . . 289
- std namespace, missing from EC++
  - and Extended EC++ . . . . . 165
- stdarg.h (library header file) . . . . . 286, 291
- stdbool.h (library header file) . . . . . 228, 286, 291
- \_\_STDC\_\_ (predefined symbol) . . . . . 281
- STDC CX\_LIMITED\_RANGE (pragma directive) . . . . . 265
- STDC FENV\_ACCESS (pragma directive) . . . . . 265
- STDC FP\_CONTRACT (pragma directive) . . . . . 265
- \_\_STDC\_VERSION\_\_ (predefined symbol) . . . . . 281
- stddef.h (library header file) . . . . . 229, 286, 291
- stderr . . . . . 102, 218
- stdin . . . . . 102
  - implementation-defined behavior in C89 (CLIB) . . . . . 338
  - implementation-defined behavior in C89 (DLIB) . . . . . 335
- stdint.h (library header file) . . . . . 286, 288
- stdio.h (library header file) . . . . . 286, 291
- stdio.h, additional C functionality . . . . . 289
- stdlib.h (library header file) . . . . . 286, 291
- stdout . . . . . 102, 218
  - implementation-defined behavior . . . . . 321
  - implementation-defined behavior in C89 (CLIB) . . . . . 338
  - implementation-defined behavior in C89 (DLIB) . . . . . 335
- Steele, Guy L. . . . . 28
- STL . . . . . 161
- strcasecmp, in string.h . . . . . 290
- strdup, in string.h . . . . . 290
- streambuf (library header file) . . . . . 287
- streams
  - implementation-defined behavior . . . . . 312
  - supported in Embedded C++ . . . . . 154
- strerror (library function)
  - implementation-defined behavior in C89 (CLIB) . . . . . 340
  - implementation-defined behavior in C89 (DLIB) . . . . . 337
- strict (compiler option) . . . . . 223
- string (library header file) . . . . . 287
- strings, supported in Embedded C++ . . . . . 154
- string.h (library header file) . . . . . 286, 291
- string.h, additional C functionality . . . . . 290
- strncasecmp, in string.h . . . . . 290
- strlen, in string.h . . . . . 290
- Stroustrup, Bjarne . . . . . 29
- strstream (library header file) . . . . . 287
- strtod (library function), configuring support for . . . . . 107
- structure types
  - alignment . . . . . 234–235
  - layout of . . . . . 234
  - packed . . . . . 235
- structures
  - accessing using a pointer . . . . . 136
  - aligning . . . . . 261
  - anonymous . . . . . 148, 171
  - implementation-defined behavior . . . . . 317
  - implementation-defined behavior in C89 . . . . . 331
  - packing and unpacking . . . . . 171
  - placing in memory type . . . . . 50
- subnormal numbers . . . . . 233
- support, technical . . . . . 194
- Sutter, Herb . . . . . 29
- symbols
  - anonymous, creating . . . . . 145
  - including in output . . . . . 262
  - listing in linker map file . . . . . 77
  - overview of predefined . . . . . 41
  - preprocessor, defining . . . . . 202
- syntax
  - command line options . . . . . 195
  - extended keywords . . . . . 48, 240–242
  - invoking compiler . . . . . 189
- system function, implementation-defined behavior . . 313, 324
- system startup
  - CLIB . . . . . 118
  - customizing . . . . . 95
  - DLIB . . . . . 93

|                                               |          |
|-----------------------------------------------|----------|
| system termination                            |          |
| CLIB                                          | 118      |
| C-SPY interface to                            | 95       |
| DLIB                                          | 94       |
| system (library function)                     |          |
| configuring support for                       | 105      |
| implementation-defined behavior in C89        | 340      |
| implementation-defined behavior in C89 (DLIB) | 337      |
| system_include (pragma directive)             | 319, 334 |
| --system_include_dir (compiler option)        | 224      |

## T

|                                                      |          |
|------------------------------------------------------|----------|
| tan (library function)                               | 284      |
| tan (library routine)                                | 107–108  |
| tanf (library routine)                               | 108–109  |
| tanl (library routine)                               | 108–109  |
| __task (extended keyword)                            | 249      |
| technical support, IAR Systems                       | 194      |
| template support                                     |          |
| in Extended EC++                                     | 154, 161 |
| missing from Embedded C++                            | 153      |
| Terminal I/O window                                  |          |
| making available (CLIB)                              | 119      |
| making available (DLIB)                              | 88       |
| not supported when                                   | 91       |
| terminal I/O, debugger runtime interface for         | 87       |
| terminal output, speeding up                         | 88       |
| termination of system. <i>See</i> system termination |          |
| termination status, implementation-defined behavior  | 323      |
| terminology                                          | 29       |
| tgmath.h (library header file)                       | 286      |
| 32-bits (floating-point format)                      | 232      |
| this (pointer)                                       | 127      |
| class memory                                         | 156      |
| referring to a class object                          | 156      |
| __TIME__ (predefined symbol)                         | 281      |
| time zone (library function)                         |          |
| implementation-defined behavior in C89               | 337, 340 |

|                                                               |               |
|---------------------------------------------------------------|---------------|
| time zone (library function), implementation-defined behavior | 324           |
| time-critical routines                                        | 121, 146, 269 |
| time.c                                                        | 106           |
| time.h (library header file)                                  | 286           |
| time32 (library function), configuring support for            | 106           |
| tips, programming                                             | 181           |
| tools icon, in this guide                                     | 30            |
| trademarks                                                    | 2             |
| transformations, compiler                                     | 177           |
| translation, implementation-defined behavior                  | 311           |
| translation, implementation-defined behavior in C89           | 327           |
| __trap (extended keyword)                                     | 57, 249       |
| trap functions                                                | 57            |
| trap vectors, specifying with pragma directive                | 267           |
| type attributes                                               | 239           |
| specifying                                                    | 266           |
| type definitions, used for specifying memory storage          | 49, 241       |
| type information, omitting                                    | 218           |
| type qualifiers                                               |               |
| const and volatile                                            | 236           |
| implementation-defined behavior                               | 318           |
| implementation-defined behavior in C89                        | 332           |
| typedefs                                                      |               |
| excluding from diagnostics                                    | 216           |
| repeated                                                      | 150           |
| type_attribute (pragma directive)                             | 49, 266       |
| type-based alias analysis (compiler transformation)           | 180           |
| disabling                                                     | 216           |
| type-safe memory management                                   | 153           |
| typographic conventions                                       | 30            |

## U

|                                 |     |
|---------------------------------|-----|
| UBROF                           |     |
| format of linkable object files | 191 |
| specifying, example of          | 37  |
| uchar.h (library header file)   | 286 |
| uintptr_t (integer type)        | 234 |

underflow errors, implementation-defined behavior . . . . 321  
 underflow range errors,  
 implementation-defined behavior in C89 . . . . . 334, 338  
 \_\_ungetchar, in stdio.h . . . . . 289  
 unions  
   anonymous . . . . . 148, 171  
   implementation-defined behavior . . . . . 317  
   implementation-defined behavior in C89 . . . . . 331  
 universal character names, implementation-defined  
 behavior . . . . . 318  
 unsigned char (data type) . . . . . 228–229  
   changing to signed char . . . . . 201  
 unsigned int (data type) . . . . . 228  
 unsigned long long (data type) . . . . . 228  
 unsigned long (data type) . . . . . 228  
 unsigned short (data type) . . . . . 228  
 \_\_user (extended keyword) . . . . . 249  
 user functions . . . . . 58  
 --use\_c++\_inline (compiler option) . . . . . 224  
 utility (STL header file) . . . . . 288

## V

variable type information, omitting in object output . . . . 218  
 variables  
   auto . . . . . 52  
   defined inside a function . . . . . 52  
   global  
     accessing . . . . . 136  
     initialization of . . . . . 71  
     placement in memory . . . . . 43  
   hints for choosing . . . . . 182  
   local. *See* auto variables  
   non-initialized . . . . . 186  
   omitting type info . . . . . 218  
   placing at absolute addresses . . . . . 175  
   placing in named segments . . . . . 175  
   static  
     placement in memory . . . . . 43

    taking the address of . . . . . 182  
     static and global, initializing . . . . . 71  
 variadic macros . . . . . 149  
 vector (pragma directive) . . . . . 57, 266  
 vector (STL header file) . . . . . 288  
 version  
   compiler subversion number . . . . . 281  
   of compiler . . . . . 281  
 version number  
   of this guide . . . . . 2  
 --vla (compiler option) . . . . . 224  
 void, pointers to . . . . . 150  
 volatile  
   and const, declaring objects . . . . . 237  
   declaring objects . . . . . 236  
   protecting simultaneously accesses variables . . . . . 184  
   rules for access . . . . . 237

## W

WAIT (assembler instruction) . . . . . 275  
 \_\_wait\_for\_interrupt (intrinsic function) . . . . . 275  
 #warning message (preprocessor extension) . . . . . 282  
 warnings . . . . . 193  
   classifying in compiler . . . . . 206  
   disabling in compiler . . . . . 217  
   exit code in compiler . . . . . 225  
 warnings icon, in this guide . . . . . 30  
 warnings (pragma directive) . . . . . 320, 334  
 --warnings\_affect\_exit\_code (compiler option) . . . . 192, 225  
 --warnings\_are\_errors (compiler option) . . . . . 225  
 wchar\_t (data type), adding support for in C . . . . . 229  
 wchar.h (library header file) . . . . . 286, 289  
 wctype.h (library header file) . . . . . 286  
 web sites, recommended . . . . . 29  
 white-space characters, implementation-defined behavior 312  
 \_\_write (library function) . . . . . 102  
   customizing . . . . . 98  
 write formatter, selecting . . . . . 116–117

|                                                   |     |
|---------------------------------------------------|-----|
| __write_array, in stdio.h . . . . .               | 289 |
| __write_buffered (DLIB library function). . . . . | 88  |

## X

|                                      |     |
|--------------------------------------|-----|
| XLINK errors                         |     |
| range error . . . . .                | 76  |
| segment too long . . . . .           | 76  |
| XLINK segment memory types . . . . . | 66  |
| xreportassert.c. . . . .             | 109 |

## Z

|                          |     |
|--------------------------|-----|
| ZINIT (segment). . . . . | 310 |
|--------------------------|-----|

# Symbols

|                                                                           |     |
|---------------------------------------------------------------------------|-----|
| __Exit (library function) . . . . .                                       | 95  |
| __exit (library function) . . . . .                                       | 95  |
| __formatted_write (library function) . . . . .                            | 115 |
| __large_write (library function) . . . . .                                | 116 |
| __medium_write (library function). . . . .                                | 116 |
| __small_write (library function) . . . . .                                | 116 |
| __adjust_return_address (intrinsic function). . . . .                     | 270 |
| __ALIGNOF__ (operator) . . . . .                                          | 148 |
| __asm (language extension) . . . . .                                      | 123 |
| __assignment_by_bitwise_copy_allowed, symbol used<br>in library . . . . . | 290 |
| __BASE_FILE__ (predefined symbol). . . . .                                | 278 |
| __BUILD_NUMBER__ (predefined symbol) . . . . .                            | 278 |
| __clear_processor_register_bit (intrinsic function). . . . .              | 270 |
| __clear_PSR_I_bit (intrinsic function). . . . .                           | 271 |
| __close (library function) . . . . .                                      | 102 |
| __code_model (runtime model attribute) . . . . .                          | 111 |
| __CODE_MODEL__ (predefined symbol). . . . .                               | 278 |
| __code, symbol used in library . . . . .                                  | 290 |
| __constrange(), symbol used in library. . . . .                           | 290 |

|                                                                             |          |
|-----------------------------------------------------------------------------|----------|
| __construction_by_bitwise_copy_allowed, symbol used<br>in library . . . . . | 290      |
| __cplusplus (predefined symbol) . . . . .                                   | 278      |
| __DATA_MODEL__ (predefined symbol) . . . . .                                | 278      |
| __data16 (extended keyword). . . . .                                        | 243      |
| __data20 (extended keyword). . . . .                                        | 244      |
| __data24 (extended keyword). . . . .                                        | 244      |
| __data32 (extended keyword). . . . .                                        | 245      |
| __DATE__ (predefined symbol) . . . . .                                      | 279      |
| __disable_interrupt (intrinsic function). . . . .                           | 271      |
| __DLIB_FILE_DESCRIPTOR (configuration symbol) . . . . .                     | 102      |
| __DOUBLE__ (predefined symbol). . . . .                                     | 279      |
| __embedded_cplusplus (predefined symbol) . . . . .                          | 279      |
| __enable_interrupt (intrinsic function) . . . . .                           | 271      |
| __enable_interrupt_wait (intrinsic function). . . . .                       | 271      |
| __exit (library function) . . . . .                                         | 95       |
| __FILE__ (predefined symbol). . . . .                                       | 279      |
| __FUNCTION__ (predefined symbol) . . . . .                                  | 152, 280 |
| __func__ (predefined symbol) . . . . .                                      | 152, 279 |
| __gets, in stdio.h. . . . .                                                 | 289      |
| __get_interrupt_state (intrinsic function) . . . . .                        | 271      |
| __get_processor_register (intrinsic function) . . . . .                     | 272      |
| __has_constructor, symbol used in library . . . . .                         | 290      |
| __has_destructor, symbol used in library . . . . .                          | 290      |
| __iar_cos_accurate (library routine). . . . .                               | 108      |
| __iar_cos_accuratef (library routine) . . . . .                             | 109      |
| __iar_cos_accuratel (library routine) . . . . .                             | 109      |
| __iar_cos_small (library routine) . . . . .                                 | 107      |
| __iar_cos_smallf (library routine). . . . .                                 | 108      |
| __iar_cos_smallll (library routine). . . . .                                | 108      |
| __iar_exp_small (library routine) . . . . .                                 | 107      |
| __iar_exp_smallf (library routine) . . . . .                                | 108      |
| __iar_exp_smallll (library routine) . . . . .                               | 108      |
| __iar_log_small (library routine) . . . . .                                 | 107      |
| __iar_log_smallf (library routine). . . . .                                 | 108      |
| __iar_log_smallll (library routine). . . . .                                | 108      |
| __iar_log10_small (library routine) . . . . .                               | 107      |
| __iar_log10_smallf (library routine). . . . .                               | 108      |
| __iar_log10_smallll (library routine). . . . .                              | 108      |
| __iar_Pow (library routine). . . . .                                        | 108      |

- `__iar_Powf` (library routine) . . . . . 109
- `__iar_Powl` (library routine) . . . . . 109
- `__iar_Pow_accurate` (library routine) . . . . . 108
- `__iar_pow_accurate` (library routine) . . . . . 108
- `__iar_Pow_accuratef` (library routine) . . . . . 109
- `__iar_pow_accuratef` (library routine) . . . . . 109
- `__iar_Pow_accuratel` (library routine) . . . . . 109
- `__iar_pow_accuratel` (library routine) . . . . . 109
- `__iar_pow_small` (library routine) . . . . . 107
- `__iar_pow_smallf` (library routine) . . . . . 108
- `__iar_pow_smallll` (library routine) . . . . . 108
- `__iar_Sin` (library routine) . . . . . 107–108
- `__iar_Sinf` (library routine) . . . . . 108–109
- `__iar_Sinl` (library routine) . . . . . 108–109
- `__iar_Sin_accurate` (library routine) . . . . . 108
- `__iar_sin_accurate` (library routine) . . . . . 108
- `__iar_Sin_accuratef` (library routine) . . . . . 109
- `__iar_sin_accuratef` (library routine) . . . . . 109
- `__iar_Sin_accuratel` (library routine) . . . . . 109
- `__iar_sin_accuratel` (library routine) . . . . . 109
- `__iar_Sin_small` (library routine) . . . . . 107
- `__iar_sin_small` (library routine) . . . . . 107
- `__iar_Sin_smallf` (library routine) . . . . . 108
- `__iar_sin_smallf` (library routine) . . . . . 108
- `__iar_Sin_smallll` (library routine) . . . . . 108
- `__iar_sin_smallll` (library routine) . . . . . 108
- `__IAR_SYSTEMS_ICC__` (predefined symbol) . . . . . 280
- `__iar_tan_accurate` (library routine) . . . . . 108
- `__iar_tan_accuratef` (library routine) . . . . . 109
- `__iar_tan_accuratel` (library routine) . . . . . 109
- `__iar_tan_small` (library routine) . . . . . 107
- `__iar_tan_smallf` (library routine) . . . . . 108
- `__iar_tan_smallll` (library routine) . . . . . 108
- `__ICCCR16C__` (predefined symbol) . . . . . 280
- `__INDEXED_ENABLED__` (predefined symbol) . . . . . 280
- `__interrupt` (extended keyword) . . . . . 57, 245
  - using in pragma directives . . . . . 267
- `__intrinsic` (extended keyword) . . . . . 246
- `__ix20` (extended keyword) . . . . . 246
- `__ix4` (extended keyword) . . . . . 246
- `__LINE__` (predefined symbol) . . . . . 280
- `__LITTLE_ENDIAN__` (predefined symbol) . . . . . 280
- `__low_level_init` . . . . . 93
  - customizing . . . . . 96
- `__lseek` (library function) . . . . . 102
- `__mac_q15` (intrinsic function) . . . . . 272
- `__mac_signed` (intrinsic function) . . . . . 273
- `__mac_unsigned` (intrinsic function) . . . . . 273
- `__memcpy_generic` (intrinsic function) . . . . . 273
- `__memory_of`
  - operator . . . . . 157
  - symbol used in library . . . . . 290
- `__memset_generic` (intrinsic function) . . . . . 273
- `__monitor` (extended keyword) . . . . . 247
- `__noadjust` (extended keyword) . . . . . 247
- `__noreturn` (extended keyword) . . . . . 248
- `__no_init` (extended keyword) . . . . . 186, 247
- `__no_operation` (intrinsic function) . . . . . 274
- `__open` (library function) . . . . . 102
- `__PRETTY_FUNCTION__` (predefined symbol) . . . . . 280
- `__printf_args` (pragma directive) . . . . . 262
- `__program_start` (label) . . . . . 93
- `__raise_exception` (intrinsic function) . . . . . 274
- `__raw` (extended keyword) . . . . . 248
- `__read` (library function) . . . . . 102
  - customizing . . . . . 98
- `__ReportAssert` (library function) . . . . . 109
- `__root` (extended keyword) . . . . . 248
- `__rt_version` (runtime model attribute) . . . . . 111
- `__scanf_args` (pragma directive) . . . . . 263
- `__segment_begin` (extended operator) . . . . . 149
- `__segment_end` (extended operators) . . . . . 149
- `__segment_size` (extended operators) . . . . . 149
- `__set_interrupt_state` (intrinsic function) . . . . . 274
- `__set_processor_register` (intrinsic function) . . . . . 274
- `__set_processor_register_bit` (intrinsic function) . . . . . 274
- `__set_PSR_I_bit` (intrinsic function) . . . . . 275
- `__STDC_VERSION__` (predefined symbol) . . . . . 281

|                                                      |         |                                                              |          |
|------------------------------------------------------|---------|--------------------------------------------------------------|----------|
| __STDC__ (predefined symbol) . . . . .               | 281     | --enable_multibytes (compiler option) . . . . .              | 209      |
| __task (extended keyword) . . . . .                  | 249     | --error_limit (compiler option) . . . . .                    | 209      |
| __TIME__ (predefined symbol) . . . . .               | 281     | --guard_calls (compiler option) . . . . .                    | 210      |
| __trap (extended keyword) . . . . .                  | 57, 249 | --header_context (compiler option) . . . . .                 | 210      |
| __ungetchar, in stdio.h . . . . .                    | 289     | --library_module (compiler option) . . . . .                 | 212      |
| __user (extended keyword) . . . . .                  | 249     | --macro_positions_in_diagnostics (compiler option) . . . . . | 212      |
| __VA_ARGS__ (preprocessor extension) . . . . .       | 145     | --mfc (compiler option) . . . . .                            | 213      |
| __wait_for_interrupt (intrinsic function) . . . . .  | 275     | --misrac (compiler option) . . . . .                         | 198      |
| __write (library function) . . . . .                 | 102     | --misrac_verbose (compiler option) . . . . .                 | 199      |
| customizing . . . . .                                | 98      | --misrac1998 (compiler option) . . . . .                     | 199      |
| __write_array, in stdio.h . . . . .                  | 289     | --misrac2004 (compiler option) . . . . .                     | 199      |
| __write_buffered (DLIB library function) . . . . .   | 88      | --module_name (compiler option) . . . . .                    | 213      |
| -D (compiler option) . . . . .                       | 202     | --no_code_motion (compiler option) . . . . .                 | 213      |
| -e (compiler option) . . . . .                       | 208     | --no_cse (compiler option) . . . . .                         | 214      |
| -f (compiler option) . . . . .                       | 210     | --no_inline (compiler option) . . . . .                      | 214      |
| -I (compiler option) . . . . .                       | 211     | --no_path_in_file_macros (compiler option) . . . . .         | 214      |
| -l (compiler option) . . . . .                       | 211     | --no_size_constraints (compiler option) . . . . .            | 215      |
| for creating skeleton code . . . . .                 | 126     | --no_static_destruction (compiler option) . . . . .          | 215      |
| -O (compiler option) . . . . .                       | 217     | --no_system_include (compiler option) . . . . .              | 215      |
| -o (compiler option) . . . . .                       | 218     | --no_typedefs_in_diagnostics (compiler option) . . . . .     | 216      |
| -r (compiler option) . . . . .                       | 203     | --no_unroll (compiler option) . . . . .                      | 216      |
| --char_is_signed (compiler option) . . . . .         | 201     | --no_warnings (compiler option) . . . . .                    | 217      |
| --char_is_unsigned (compiler option) . . . . .       | 201     | --no_wrap_diagnostics (compiler option) . . . . .            | 217      |
| --clib (compiler option) . . . . .                   | 201     | --omit_types (compiler option) . . . . .                     | 218      |
| --code_model (compiler option) . . . . .             | 202     | --only_stdout (compiler option) . . . . .                    | 218      |
| --c89 (compiler option) . . . . .                    | 200     | --output (compiler option) . . . . .                         | 218      |
| --data_model (compiler option) . . . . .             | 203     | --predef_macro (compiler option) . . . . .                   | 219      |
| --debug (compiler option) . . . . .                  | 203     | --preinclude (compiler option) . . . . .                     | 219      |
| --dependencies (compiler option) . . . . .           | 203     | --preprocess (compiler option) . . . . .                     | 220      |
| --diagnostics_tables (compiler option) . . . . .     | 206     | --relaxed_fp (compiler option) . . . . .                     | 220      |
| --diag_error (compiler option) . . . . .             | 204     | --remarks (compiler option) . . . . .                        | 221      |
| --diag_remark (compiler option) . . . . .            | 205     | --require_prototypes (compiler option) . . . . .             | 221      |
| --diag_suppress (compiler option) . . . . .          | 205     | --segment (compiler option) . . . . .                        | 222      |
| --diag_warning (compiler option) . . . . .           | 206     | --silent (compiler option) . . . . .                         | 223      |
| --discard_unused_publics (compiler option) . . . . . | 206     | --strict (compiler option) . . . . .                         | 223      |
| --dlib (compiler option) . . . . .                   | 207     | --system_include_dir (compiler option) . . . . .             | 224      |
| --dlib_config (compiler option) . . . . .            | 207     | --use_c++_inline (compiler option) . . . . .                 | 224      |
| --ec++ (compiler option) . . . . .                   | 208     | --vla (compiler option) . . . . .                            | 224      |
| --eec++ (compiler option) . . . . .                  | 209     | --warnings_affect_exit_code (compiler option) . . . . .      | 192, 225 |

--warnings\_are\_errors (compiler option) . . . . . 225  
 ?C\_EXIT (assembler label) . . . . . 119  
 ?C\_GETCHAR (assembler label) . . . . . 119  
 ?C\_PUTCHAR (assembler label) . . . . . 119  
 @ (operator)  
     placing at absolute address . . . . . 174  
     placing in segments . . . . . 175  
 #include files, specifying . . . . . 190, 211  
 #warning message (preprocessor extension) . . . . . 282  
 %Z replacement string,  
 implementation-defined behavior . . . . . 324

## Numerics

16-bit pointers, accessing memory . . . . . 45  
 32-bit pointers, accessing memory . . . . . 45  
 32-bits (floating-point format) . . . . . 232  
 64-bit data types, avoiding . . . . . 169  
 64-bits (floating-point format) . . . . . 232