

IAR C/C++ Compiler

Reference Guide

for the Renesas

R32C/I00 Microcomputer Family

COPYRIGHT NOTICE

Copyright © 2007–2009 IAR Systems AB.

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

DISCLAIMER

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

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

TRADEMARKS

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Renesas is a registered trademark of Renesas Technology Corporation. R32C/100 is a trademark of Renesas Technology Corporation.

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

EDITION NOTICE

Second edition: January 2009

Part number: CR32C-2

This guide applies to version 1.x of IAR Embedded Workbench® for R32C.

Internal reference: R7, 5.4, IJOA

Brief contents

Tables	xxi
Preface	xxiii
Part 1. Using the compiler	1
Getting started	3
Data storage	11
Functions	21
Placing code and data	27
The DLIB runtime environment	41
Assembler language interface	71
Using C++	91
Efficient coding for embedded applications	99
Part 2. Reference information	117
External interface details	119
Compiler options	125
Data representation	153
Compiler extensions	163
Extended keywords	173
Pragma directives	185
Intrinsic functions	199
The preprocessor	211
Library functions	217

Segment reference	225
Implementation-defined behavior	239
Index	251

Contents

Tables	xxi
Preface	xxiii
Who should read this guide	xxiii
How to use this guide	xxiii
What this guide contains	xxiv
Other documentation	xxv
Further reading	xxv
Document conventions	xxvi
Typographic conventions	xxvi
Naming conventions	xxvii
 Part I. Using the compiler	 1
Getting started	3
IAR language overview	3
Supported R32C/I00 devices	4
Building applications—an overview	4
Compiling	4
Linking	4
Basic settings for project configuration	5
Data model	6
Code model	6
Size of double floating-point type	6
Optimization for speed and size	7
Runtime environment	7
Special support for embedded systems	8
Extended keywords	8
Pragma directives	9
Predefined symbols	9
Special function types	9
Accessing low-level features	9

Data storage	11
Introduction	11
Different ways to store data	11
Data models	12
Specifying a data model	12
Memory types	13
Data16	14
Data24	14
Data32	14
Sbdata16	14
Sbdata24	14
Using data memory attributes	14
Structures and memory types	16
More examples	16
C++ and memory types	17
Auto variables—on the stack	17
The stack	18
Dynamic memory on the heap	19
Functions	21
Function-related extensions	21
Code models and memory attributes for function storage	21
Using function memory attributes	22
Primitives for interrupts, concurrency, and OS-related programming	22
Interrupt functions	23
Monitor functions	24
C++ and special function types	26
Placing code and data	27
Segments and memory	27
What is a segment?	27
Placing segments in memory	28
Customizing the linker command file	28

Data segments	31
Static memory segments	31
The internal data stack	34
The interrupt stack	35
The heap	35
Located data	37
Code segments	37
Startup code	37
Normal code	37
Interrupt vectors	38
C++ dynamic initialization	38
Verifying the linked result of code and data placement	38
Segment too long errors and range errors	38
Linker map file	39
The DLIB runtime environment	41
Introduction to the runtime environment	41
Runtime environment functionality	41
Library selection	42
Situations that require library building	43
Library configurations	43
Debug support in the runtime library	44
Using a prebuilt library	44
Customizing a prebuilt library without rebuilding	46
Choosing formatters for printf and scanf	47
Choosing printf formatter	47
Choosing scanf formatter	48
Overriding library modules	49
Building and using a customized library	50
Setting up a library project	51
Modifying the library functionality	51
Using a customized library	51
System startup and termination	52
System startup	52

System termination	54
Customizing system initialization	55
__low_level_init	55
Modifying the file cstartup.s53	55
Standard streams for input and output	56
Implementing low-level character input and output	56
Configuration symbols for printf and scanf	58
Customizing formatting capabilities	59
File input and output	59
Locale	60
Locale support in prebuilt libraries	60
Customizing the locale support	60
Changing locales at runtime	61
Environment interaction	62
Signal and raise	63
Time	63
Strtod	63
Assert	64
Hardware support	64
Floating-point implementation	64
C-SPY runtime interface	65
Low-level debugger runtime interface	66
The debugger terminal I/O window	66
Checking module consistency	67
Runtime model attributes	67
Using runtime model attributes	68
Predefined runtime attributes	68
User-defined runtime model attributes	69
Assembler language interface	71
Mixing C and assembler	71
Intrinsic functions	71
Mixing C and assembler modules	72
Inline assembler	73

Calling assembler routines from C	74
Creating skeleton code	74
Compiling the code	75
Calling assembler routines from C++	76
Calling convention	77
Function declarations	78
Using C linkage in C++ source code	78
Preserved versus scratch registers	79
Function entrance	79
Function exit	82
Restrictions for special function types	83
Examples	83
Function directives	84
Calling functions	85
Assembler instructions used for calling functions	85
Memory access methods	86
The data16 memory access method	87
The data24 memory access method	87
The data32 memory access method	87
The sbdata16 memory access method	88
The sbdata24 memory access method	88
Call frame information	88
Using C++	91
Overview	91
Standard Embedded C++	91
Extended Embedded C++	92
Enabling C++ support	92
Feature descriptions	93
Classes	93
Functions	94
Templates	94
Variants of casts	95
Mutable	95

Namespace	95
The STD namespace	95
Using interrupts and EC++ destructors	95
C++ language extensions	96
Efficient coding for embedded applications	99
Selecting data types	99
Using efficient data types	99
Floating-point types	100
Casting a floating-point value to an integer	101
Alignment of elements in a structure	101
Anonymous structs and unions	101
Controlling data and function placement in memory	103
Data placement at an absolute location	104
Data and function placement in segments	105
Controlling compiler optimizations	106
Scope for performed optimizations	107
Optimization levels	107
Speed versus size	108
Fine-tuning enabled transformations	109
Writing efficient code	111
Saving stack space and RAM memory	112
Function prototypes	112
Integer types and bit negation	113
Protecting simultaneously accessed variables	113
Accessing special function registers	114
Non-initialized variables	115
Part 2. Reference information	117
External interface details	119
Invocation syntax	119
Compiler invocation syntax	119
Passing options	119

Environment variables	120
Include file search procedure	120
Compiler output	121
Diagnostics	122
Message format	123
Severity levels	123
Setting the severity level	124
Internal error	124
Compiler options	125
Options syntax	125
Types of options	125
Rules for specifying parameters	125
Summary of compiler options	128
Descriptions of options	130
--align_func	130
--char_is_signed	131
--code_model	131
-D	131
--data_model	132
--debug, -r	132
--dependencies	133
--diag_error	134
--diag_remark	134
--diag_suppress	134
--diag_warning	135
--diagnostics_tables	135
--discard_unused_publics	136
--dlib_config	136
--double	137
-e	137
--ec++	137
--eec++	138
--enable_multibytes	138

--error_limit	138
-f	139
--fp_model	139
--header_context	140
-I	140
-l	140
--library_module	141
--mfc	141
--migration_preprocessor_extensions	142
--module_name	142
--no_code_motion	143
--no_cross_call	143
--no_cse	143
--no_inline	144
--no_path_in_file_macros	144
--no_tbaa	145
--no_typedefs_in_diagnostics	145
--no_unroll	146
--no_warnings	146
--no_wrap_diagnostics	146
-O	147
-o, --output	147
--omit_types	148
--only_stdout	148
-o, --output	148
--predef_macros	149
--preinclude	149
--preprocess	149
--public_equ	150
-r, --debug	150
--remarks	150
--require_prototypes	151
--silent	151
--strict_ansi	151

--warnings_affect_exit_code	152
--warnings_are_errors	152
Data representation	153
Alignment	153
Alignment on the R32C/100 microcomputer	153
Basic data types	154
Integer types	154
Floating-point types	156
Pointer types	157
Function pointers	157
Data pointers	158
Casting	158
Structure types	158
Alignment	159
General layout	159
Packed structure types	159
Type qualifiers	161
Declaring objects volatile	161
Declaring objects const	162
Data types in C++	162
Compiler extensions	163
Compiler extensions overview	163
Enabling language extensions	164
C language extensions	164
Important language extensions	164
Useful language extensions	166
Minor language extensions	169
Extended keywords	173
General syntax rules for extended keywords	173
Type attributes	173
Object attributes	176
Summary of extended keywords	177

Descriptions of extended keywords	177
__code24	177
__code32	178
__data16	178
__data24	179
__data32	179
__fast_interrupt	179
__interrupt	180
__intrinsic	180
__monitor	180
__no_init	181
__noreturn	181
__packed	181
__root	182
__sbddata16	182
__sbddata24	183
__task	183
Pragma directives	185
Summary of pragma directives	185
Descriptions of pragma directives	186
bitfields	186
constseg	187
data_alignment	187
dataseg	188
diag_default	188
diag_error	189
diag_remark	189
diag_suppress	189
diag_warning	190
include_alias	190
inline	191
language	191
location	192

message	192
object_attribute	193
optimize	193
pack	194
__printf_args	195
required	195
rtmodel	196
__scanf_args	196
segment	197
type_attribute	197
vector	198
Intrinsic functions	199
Summary of intrinsic functions	199
Descriptions of intrinsic functions	201
__break	201
__delay_cycles	201
__disable_interrupt	201
__enable_interrupt	201
__exchange_byte	201
__exchange_long	202
__exchange_word	202
__get_DCR_register	202
__get_DCT_register	202
__get_DDA_register	202
__get_DDR_register	202
__get_DMD_register	203
__get_DSA_register	203
__get_DSR_register	203
__get_interrupt_level	203
__get_interrupt_state	203
__get_interrupt_table	204
__get_VCT_register	204
__illegal_opcode	204

__interrupt_on_overflow	204
__load_context	204
__low_level_init	205
__no_operation	205
__RMPA_B	205
__RMPA_L	205
__RMPA_W	205
__ROUND	206
__set_DCR_register	206
__set_DCT_register	206
__set_DDA_register	206
__set_DDR_register	206
__set_DMD_register	206
__set_DSA_register	207
__set_DSR_register	207
__set_interrupt_level	207
__set_interrupt_state	207
__set_interrupt_table	207
__set_VCT_register	207
__SIN	208
__software_interrupt	208
__SOUT	208
__STOP	208
__store_context	208
__wait_for_interrupt	209

The preprocessor	211
------------------------	-----

Overview of the preprocessor	211
---	-----

Descriptions of predefined preprocessor symbols	212
--	-----

Descriptions of miscellaneous preprocessor extensions	214
--	-----

NDEBUG	214
_Pragma()	215
#warning message	215
__VA_ARGS__	215

Library functions	217
Introduction	217
Header files	217
Library object files	217
Reentrancy	218
IAR DLIB Library	218
C header files	219
C++ header files	220
Library functions as intrinsic functions	222
Added C functionality	222
Segment reference	225
Summary of segments	225
Descriptions of segments	226
CHECKSUM	227
CODE24	227
CODE32	227
CSTACK	228
CSTART	228
DATA16_AC	228
DATA16_AN	228
DATA16_C	229
DATA16_I	229
DATA16_ID	229
DATA16_N	230
DATA16_Z	230
DATA24_AC	230
DATA24_AN	230
DATA24_C	231
DATA24_I	231
DATA24_ID	231
DATA24_N	232
DATA24_Z	232
DATA32_AC	232

DATA32_AN	232
DATA32_C	233
DATA32_I	233
DATA32_ID	233
DATA32_N	234
DATA32_Z	234
DIFUNCT	234
HEAP	234
INTVEC	235
ISTACK	235
NMIVEC	235
SBDATA16_I	236
SBDATA16_ID	236
SBDATA16_N	236
SBDATA16_Z	237
SBDATA24_I	237
SBDATA24_ID	237
SBDATA24_N	238
SBDATA24_Z	238
Implementation-defined behavior	239
Descriptions of implementation-defined behavior	239
Translation	239
Environment	240
Identifiers	240
Characters	240
Integers	242
Floating point	242
Arrays and pointers	243
Registers	243
Structures, unions, enumerations, and bitfields	243
Qualifiers	244
Declarators	244
Statements	244

Preprocessing directives 244

IAR DLIB Library functions 246

Index 251

Tables

1: Typographic conventions used in this guide	xxvi
2: Naming conventions used in this guide	xxvii
3: Command line options for specifying library and dependency files	8
4: Data model characteristics	12
5: Memory types and their corresponding memory attributes	15
6: Code models	22
7: Function memory attributes	22
8: XLINK segment memory types	28
9: Memory layout of a target system (example)	29
10: Memory types with corresponding segment groups	32
11: Segment name suffixes	32
12: Library configurations	43
13: Levels of debugging support in runtime libraries	44
14: Prebuilt libraries	45
15: Customizable items	46
16: Formatters for printf	47
17: Formatters for scanf	48
18: Descriptions of printf configuration symbols	58
19: Descriptions of scanf configuration symbols	58
20: Low-level I/O files	59
21: Functions with special meanings when linked with debug info	65
22: Example of runtime model attributes	68
23: Predefined runtime model attributes	68
24: Registers used for passing parameters	80
25: Registers used for returning values	83
26: Specifying the size of a memory address in assembler instructions	86
27: Call frame information resources defined in a names block	89
28: Compiler optimization levels	107
29: Compiler environment variables	120
30: Error return codes	122
31: Compiler options summary	128

32: Integer types	154
33: Floating-point types	156
34: Extended keywords summary	177
35: Pragma directives summary	185
36: Intrinsic functions summary	199
37: Predefined symbols	212
38: Traditional standard C header files—DLIB	219
39: Embedded C++ header files	220
40: Additional Embedded C++ header files—DLIB	220
41: Standard template library header files	221
42: New standard C header files—DLIB	221
43: Segment summary	225
44: Message returned by strerror()—IAR DLIB library	249

Preface

Welcome to the IAR C/C++ Compiler for R32C Reference Guide. 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 R32C/100 microcomputer and need detailed reference information on how to use the compiler. You should have working knowledge of:

- The architecture and instruction set of the R32C/100 microcomputer. Refer to the documentation from Renesas for information about the R32C/100 microcomputer
- 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 R32C, 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 *IAR Embedded Workbench® IDE User Guide*. This guide contains a product overview, tutorials that can help you get started, conceptual and user information about the IDE and the IAR C-SPY® Debugger, and corresponding reference information.

What this guide contains

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

Part 1. Using the 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 command 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.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *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.
- *Compiler options* 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.
- *Compiler extensions* gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.

- *Extended keywords* gives reference information about each of the R32C-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 R32C-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* describes how the compiler handles the implementation-defined areas of the C language standard.

Other documentation

The complete set of IAR Systems development tools for the R32C/100 microcomputer is described in a series of guides. For information about:

- Using the IDE and the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the R32C IAR Assembler, refer to the *R32C IAR Assembler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library functions, refer to the online help system
- Using the MISRA-C:1998 rules or the MISRA-C:2004 rules, refer to the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*, respectively.

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

FURTHER READING

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

- 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.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall. [The later editions describe the ANSI C standard.]
- 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.]
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.

We recommend that you visit these web sites:

- The Renesas web site, www.renesas.com, contains information and news about the R32C/100 microcomputers.
- The IAR Systems web site, www.iar.com, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, www.caravan.net/ec2plus, contains information about the Embedded C++ standard.

Document conventions

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

When referring to a directory in your product installation, for example `r32c\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 5.n\r32c\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
<code>computer</code>	<ul style="list-style-type: none">• Source code examples and file paths.• Text on the command line.• Binary, hexadecimal, and octal numbers.
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a command.
<code>a b c</code>	Alternatives in a command.

Table 1: Typographic conventions used in this guide





Style	Used for
{a b c}	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none">• A cross-reference within this guide or to another guide.• Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

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

NAMING CONVENTIONS

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

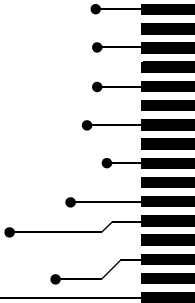
Brand name	Generic term
IAR Embedded Workbench® for R32C	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for R32C	the IDE
IAR C-SPY® Debugger for R32C	C-SPY, the debugger
IAR C/C++ Compiler™ for R32C	the compiler
IAR Assembler™ for R32C	the assembler
IAR XLINK™ Linker	XLINK, the linker
IAR XAR Library builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide

Part I. Using the compiler

This part of the *IAR C/C++ Compiler for R32C Reference Guide* includes these chapters:

- Getting started
- Data storage
- Functions
- Placing code and data
- The DLIB runtime environment
- Assembler language interface
- 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 R32C/I00 microcomputer. 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 R32C:

- C, the most widely used high-level programming language in the embedded systems industry. Using the R32C IAR C/C++ Compiler, you can build freestanding applications that follow the standard ISO 9899:1990. This standard is commonly known as ANSI C.
- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. IAR Systems supports two levels of the C++ language:
 - Embedded C++ (EC++), a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
 - IAR Extended Embedded C++, with additional features such as full template support, multiple inheritance, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *R32C IAR Assembler Reference Guide*.

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

Supported R32C/100 devices

The IAR C/C++ Compiler for R32C supports all devices based on the standard Renesas R32C/100 microcomputer. Hardware floating-point units (FPUs) are also supported.

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 command 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 *IAR Embedded Workbench® IDE User Guide*.

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r53` using the default settings:

```
iccr32c myfile.c
```

You must also specify some critical options, see *Basic settings for project configuration*, page 5.

LINKING

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

- Several object files and possibly certain libraries

- The standard library containing the runtime environment and the standard language functions
- A program start label
- A linker command 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.r53 myfile2.r53 -s __program_start -f lnkr32c.xcl
dlr32cfhfn.r53 -o aout.a53 -r
```

In this example, `myfile.r53` and `myfile2.r53` are object files, `lnkr32c.xcl` is the linker command file, and `dlr32cfhfn.r53` 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 Intel-extended.)

Basic settings for project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler generate the best code for the R32C/100 device you are using. You can specify the options either from the command line interface or in the IDE.

The basic settings are:

- Data model
- Code model
- Floating-point model
- Size of double floating-point type
- Optimization settings
- Runtime environment.

In addition to these settings, many other options and settings can fine-tune the result even further. For details about how to set options and for a list of all available options,

see the chapters *Compiler options* and the *IAR Embedded Workbench® IDE User Guide*, respectively.

DATA MODEL

One of the characteristics of the R32C/100 microcomputer 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. However, you can use memory attributes to override the default access method for each individual variable.

The following data models are supported:

- The *Near* data model can access up to 32 Kbytes of RAM memory
- The *Far* data model can access up to 8 Mbytes of RAM memory
- The *Huge* data model can access up to 4 Gbytes of RAM memory.

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 compiler supports *code models* that you can set on file- or function-level to control which function calls are generated, which determines the size of the linked application. The following code models are available:

- The *Far* code model has an upper limit of 8 Mbytes of code
- The *Huge* code model can access the entire 32-bit address space.

For detailed information about the code models, see the chapter *Functions*.

SIZE OF DOUBLE FLOATING-POINT TYPE

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE754-compliant format. If you use the compiler option `--double={32|64}`, you can choose whether data declared as `double` should be represented with 32 bits or 64 bits. The data type `float` is always represented using 32 bits.

If there is a hardware floating-point unit (FPU) available, you can choose between using the FPU's own floating-point instructions, which do not comply with the IEEE-754 standard, and using library functions that are standards-compliant but slower. See *Floating-point implementation*, page 64.

OPTIMIZATION FOR SPEED AND SIZE

The compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common 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 details about compiler optimizations and for more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

RUNTIME ENVIRONMENT

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

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

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

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

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



Choosing a runtime library in the IDE

To choose a library, choose **Project>Options**, and click the **Library Configuration** tab in the **General Options** category. Choose the appropriate library from the **Library** drop-down menu.

Note that for the DLIB library there are two different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 43, for more information.

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



Choosing runtime environment from the command line

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

Command line	Description
<code>-I r32c\inc</code>	Specifies the include path to device-specific I/O definition files.
<code>libraryfile.r53</code>	Specifies the library object file
<code>--dlib_config</code> <code>C:\...\configfile.h</code>	Specifies the library configuration file

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

For a list of all prebuilt library object files, see Table 14, *Prebuilt libraries*, page 45. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 47
- The size of the stack and the heap, see *The internal data stack*, page 34, and *The heap*, page 35, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the R32C/100 microcomputer.

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 137 for additional information.

For detailed descriptions of the extended keywords, see the chapter *Extended keywords*.

PRAGMA DIRECTIVES

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

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

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

PREDEFINED SYMBOLS

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

For detailed descriptions of the predefined symbols, see the chapter *The preprocessor*.

SPECIAL FUNCTION TYPES

The special hardware features of the R32C/100 microcomputer are supported by the compiler's special function types: interrupt, monitor, and task. You can write a complete application without having to write any of these functions in assembler language.

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 22.

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 71.

Data storage

This chapter gives a brief introduction to the memory layout of the R32C/100 microcomputer 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 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 R32C IAR C/C++ Compiler supports R32C/100 devices that have continuous memory ranging from 0x00000000 to 0xFFFFFFFF. Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

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. To read more about this, see *Memory types*, page 13.

DIFFERENT WAYS TO STORE DATA

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

- Auto variables.

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

- Global variables and local variables declared `static`.

In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Data models*, page 12 and *Memory types*, page 13.

- Dynamically allocated data.
An application can allocate data on the *heap*, where the data it 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 19.

Data models

Technically, the data model specifies the default memory type. This means that the data model controls the default placement of static and global variables.

Note: The default placement of constant data is controlled by the *code* model, see *Code models and memory attributes for function storage*, page 21.

The data model only specifies the default memory type. It is possible to override this for individual variables. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 14.

SPECIFYING A DATA MODEL

Three data models are implemented: Near, Far, and Huge. These models are controlled by the `--data_model` option. Each model has a default memory type. If you do not specify a data model option, the compiler will use the Far 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 by explicitly specifying a memory attribute, using either keywords or the `#pragma type_attribute` directive.

This table summarizes the different data models:

Data model	Default memory attribute	Placement of variable data
Near	<code>__data16</code>	0x0–0x7FFF, 0xFFFF8000–0xFFFFFFFF
Far	<code>__data24</code>	0x0–0x7FFFFFFF, 0xFF800000–0xFFFFFFFF
Huge	<code>__data32</code>	0x0–0xFFFFFFFF

Table 4: Data model characteristics



See the *IAR Embedded Workbench® IDE User 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 132.

The Near data model

The Near data model places variables in the lowest or highest 32 Kbytes of memory. This is the only memory type that can be accessed using 2-byte direct addressing.

The Far data model

The Far data model places variables in the lowest or highest 8 Mbytes of memory, using 3-byte direct addressing. It generates slightly more code than 2-byte-addressing using the Near data model, but it is more flexible.

The Huge data model

The Huge data model can place variables anywhere in memory, using indirect addressing. This is very convenient and flexible, but significantly less efficient compared to the Near and Far data models.

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—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 86.

DATA16

The data16 memory consists of the low 32 Kbytes and the high 32 Kbytes of memory. In hexadecimal notation, this is the address ranges 0x0–0x7FFF and 0xFFFF8000–0xFFFFFFFF.

A data16 object can only be placed in data16 memory, and the size of such an object is limited to 32 Kbytes-1. By using objects of this type, the code generated by the compiler to access them is minimized. This means a smaller memory footprint for the application.

DATA24

The data24 memory consists of the memory in the ranges 0x0–0x7FFFFF and 0xFF800000–0xFFFFFFFF.

The drawback of the data24 memory type is that the code generated to access the memory is slightly larger than data16 memory.

DATA32

Using this memory type, you can place the data objects anywhere in memory. The drawback is that it can only use indirect addressing, which uses more processor registers and might force local variables to be stored on the stack rather than being allocated in registers. The resulting code is significantly larger and slower.

SBDATA16

The sbdata16 memory consists of the first 64 Kbytes of memory after the SB (static base) register. In hexadecimal notation, this is the address range $SB + (0x0-0xFFFF)$.

SBDATA24

The sbdata24 memory consists of the first 16 Mbytes of memory after the SB (static base) register. In hexadecimal notation, this is the address range $SB + (0x0-0xFFFFFFFF)$.

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, 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.

Note: Even though the default placement of constant data is controlled by the *code* model, *data* memory attributes are used for overriding the default placement.

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

Memory type	Keyword	Address range for variables and constants	Default in data model
Data16	<code>__data16</code>	0x0-0x7FFF 0xFFFF8000-0xFFFFFFFF	Near
Data24	<code>__data24</code>	0x0-0x7FFFFF 0xFF800000-0xFFFFFFFF	Far
Data32	<code>__data32</code>	0x0-0xFFFFFFFF	Huge
Sbdata16	<code>__sbdata16</code>	SB + 0x0-0xFFFF	—
Sbdata24	<code>__sbdata24</code>	SB + 0x0-0xFFFFF	—

Table 5: Memory types and their corresponding memory attributes

All data pointers are 32 bits.

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 137 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 177.

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 173.

The following declarations place the variable `i` and `j` in data24 memory. The variables `k` and `l` will also be placed in data24 memory. The position of the keyword does not have any effect in this case:

```
__data24 int i, j;
int __data24 k, l;
```

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

In addition to the rules presented here—to place the keyword directly in the code—the directive `#pragma type_attribute` can 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:

```
typedef char __data24 Byte;
Byte b;

and

__data24 char b;
```

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 `data24` memory.

```
struct MyStruct
{
    int alpha;
    int beta;
};
__data24 struct MyStruct gamma;
```

This declaration is incorrect:

```
struct MySecondStruct
{
    int blue;
    __data24 int green;    /* Error! */
};
```

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. It makes no difference whether the memory attribute is placed before or after the data type.

To read the following examples, start from the left and add one qualifier at each step

<code>int a;</code>	A variable defined in default memory determined by the data model in use.
<code>int __data16 b;</code>	A variable in <code>data16</code> memory.

```
__data24 int c;
```

A variable in data24 memory.

```
int * d;
```

A pointer stored in default memory. The pointer points to an integer in default memory.

C++ and memory types

A C++ class object is placed in one memory type, in the same way as for normal C structures. However, the class members that are considered to be part of the object are the non-static member variables. The static member variables can be placed individually in any kind of memory.

Remember, in C++ there is only one instance of each static member variable, regardless of the number of class objects.

Also note that when calling class methods, the `this` pointer uses the default pointer type.

Example

In the example below, an object, named `delta`, of the type `MyClass` is defined in data16 memory. The class contains a static member variable that is stored in data24 memory.

```
// The class declaration (placed in a header file):
class MyClass
{
public:
    int alpha;
    int beta;

    __data24 static int gamma;
};

// Definitions needed (should be placed in a source file):
__data24 int MyClass::gamma;

// A variable definition:
__data16 MyClass delta;
```

Auto variables—on the stack

Variables that are defined inside a function—and not declared static—are named *auto variables* by the C standard. A few of these variables are placed in processor registers; the rest are placed on the stack. From a semantic point of view, this is equivalent. The

main differences are that accessing registers is faster, and that less memory is required compared to when variables are located on the stack.

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

THE STACK

The stack can contain:

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

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

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

Advantages

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

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

Potential problems

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

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

```
int * MyFunction()
{
    int x;
    ... do something ...
    return &x;
}
```

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

Dynamic memory on the heap

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

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

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

Potential problems

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

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

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

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

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

Functions

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

Function-related extensions

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

- Control the storage of functions in memory
- Use primitives for interrupts, concurrency, and OS-related programming
- Facilitate function optimization
- Access hardware features.

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

For more information about optimizations, see *Writing efficient code*, page 111. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models and memory attributes for function storage

By means of *code models*, the compiler supports placing functions and constant literals in a default part of memory, or in other words, use a default address size for the functions and constants. Technically, the code models control the following:

- The possible memory range for storing the function or constant
- The maximum module size
- The maximum application size
- The default memory attribute.

The compiler supports two code models. If you do not specify a code model, the compiler will use the Far code model as default. Your project can only use one code model at a time, and the same model must be used by all user modules and all library modules.

These code models are available:

Code model	Address range for placing functions and constant data
Far (default)	0xFF800000-0xFFFFFFFF
Huge	0x0-0xFFFFFFFF

Table 6: Code models



See the *IAR Embedded Workbench® IDE User Guide* for information about specifying a code model in the IDE.



Use the `--code_model` option to specify the code model for your project; see `--code_model`, page 131.

USING FUNCTION MEMORY ATTRIBUTES

It is possible to override the default placement for individual functions. Use the appropriate *function memory attribute* to specify this. These attributes are available:

Function memory attribute	Address range	Default in code model	Description
<code>__code24</code>	0xFF800000-0xFFFFFFFF	Far	The function can be placed in the high 8 Mbytes of memory.
<code>__code32</code>	0x0-0xFFFFFFFF	Huge	The function can be placed anywhere in memory.

Table 7: Function memory attributes

For detailed syntax information and for detailed information about each attribute, see the chapter *Extended keywords*.

Note: Even though the default placement of constant data is controlled by the *code* model, *data* memory attributes are used for overriding the default placement of constants.

Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for R32C provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

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

```
__get_interrupt_level, __set_interrupt_level,
__wait_for_interrupt, __interrupt_on_overflow,
__software_interrupt, __get_VCT_register, __set_VCT_register,
__get_interrupt_table, and __set_interrupt_table.
```

INTERRUPT FUNCTIONS

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

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

The R32C/100 microcomputer 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 R32C/100 microcomputer documentation from the chip manufacturer. The interrupt vector is the offset into the interrupt vector table. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors. For the R32C/100 microcomputer, the interrupt vector table always starts at the address 0xFFFFFDC for non-maskable interrupts and at the base of the `INTB` register for all other interrupts.

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

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

```
#pragma vector=TIMER_A0 /* Symbol defined in I/O header file */
__interrupt void my_interrupt_routine(void)
{
    /* Do something */
}
```

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

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. See the chip manufacturer's documentation for more information about the interrupt vector table.

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 reference information, see `__monitor`, page 180.



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 semaphore is implemented using one static variable and two monitor functions. 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.

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

/* get_lock -- Try to lock the lock.
 * Return 1 on success and 0 on failure. */

__monitor int get_lock(void)
{
    if (the_lock == 0)
    {
        /* Success, we managed to lock the lock. */
        the_lock = 1;
        return 1;
    }
    else
    {
        /* Failure, someone else has locked the lock. */
        return 0;
    }
}

/* release_lock -- Unlock the lock. */

__monitor void release_lock(void)
{
    the_lock = 0;
}
```

This is an example of a program fragment that uses the semaphore:

```
void my_program(void)
{
    if (get_lock())
    {
        /* ... Do something ... */

        /* When done, release the lock. */
        release_lock();
    }
}
```

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>

volatile long tick_count = 0;

/* Class for controlling critical blocks */
class Mutex
{
public:
    Mutex ()
    {
        _state = __get_interrupt_state();
        __disable_interrupt();
    }

    ~Mutex ()
    {
        __set_interrupt_state(_state);
    }

private:
    __istate_t _state;
};

void f()
{
```

```

static long next_stop = 100;
extern void do_stuff();
long tick;

/* A critical block */
{
    Mutex m;
    /* Read volatile variable 'tick_count' in a safe way
       and put the value in a local variable */

    tick = tick_count;
}

if (tick >= next_stop)
{
    next_stop += 100;
    do_stuff();
}
}

```

C++ AND SPECIAL FUNCTION TYPES

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

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 command 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 are 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 logical entity containing a piece of data or code that should be mapped to a physical location in memory. Each segment consists of many *segment parts*. Normally, each function or variable with static storage duration is placed in a segment part. A segment part is the smallest linkable unit, which allows the linker to include only those units that are referred to. The segment could be placed either in RAM or in ROM. Segments that are placed in RAM do not have any content, they only occupy space.

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

At compile time, the compiler assigns each segment its contents. The IAR XLINK Linker is responsible for placing the segments in the physical memory range, in accordance with the rules specified in the linker command file. Ready-made linker command files are provided, but, if necessary, they can be easily 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.

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

Segment memory type

XLINK assigns a segment memory type to each of the segments. In some cases, the individual segments have the same name as the segment memory type they belong to, for example CODE. Make sure not to confuse the individual segment names with the segment memory types 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 8: XLINK segment memory types

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

For more details about 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 command 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 derivatives, just rebuild the code with the appropriate linker command file.

In particular, the linker command file specifies:

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

This section describes the methods for placing the segments in memory, which means that you must customize the linker command file to suit the memory layout of your target system. For showing the methods, fictitious examples are used.

CUSTOMIZING THE LINKER COMMAND FILE

The `config` directory contains ready-made linker command 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 a supplied linker command 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
0x00000–0x5FFFF	RAM
0x60000–0x7FFFF	ROM
0xFFE00000–0xFFFFFFFF	ROM

Table 9: 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 command file is to verify that your application code and data do not cross the memory range boundaries, which would lead to application failure.

Remember not to change the original file. We recommend that you make a copy in the working directory, and modify the copy instead.

The contents of the linker command file

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

- The CPU used:
`-cr32c`
This specifies your target microcomputer.
- Definitions of constants used in the file. These are defined using the XLINK option `-D`.
- The placement directives (the largest part of the linker command 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 command file, all numbers are specified in hexadecimal format. However, neither the prefix `0x` nor the suffix `h` is used.

Note: The supplied linker command files include comments explaining the contents.

See the *IAR Linker and Library Tools Reference Guide* for more details.

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 0xFFFF52000-0xFFFF5CFFF.

```
-Z (CONST) MYSEGMENTA, MYSEGMENTB=FFF52000-FFF5CFFF
```

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=FFF52000-FFF5CFFF
-Z (CODE) MYCODE
```

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

```
-Z (CONST) MYSMALLSEGMENT=FFF52000-FFF520FF
-Z (CONST) MYLARGESEGMENT=FFF52000-FFF5CFFF
```



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, 0x30000-0x31000
```

If your application has an additional RAM area in the memory range 0x4F000-0x4F7FF, you can simply add that to the original definition:

```
-P (DATA) MYDATA=0-1FFF, 4F000-4F7FF, 0x30000-0x31000
```

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*—must be placed using `-Z`.

Symbols for available memory areas

To make things easier, the start and end addresses of the memory areas available for your application are defined as symbols in the linker command file:

```
// Memory areas available for the application
-D_USER_RAM_BEGIN=00000400
-D_USER_RAM_END=0003FFFF
-D_USER_ROM_BEGIN=FFE00000
-D_USER_ROM_END=FFFFFFFF
-D_SB_START=1000
```

Data segments

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

To understand how the data segments work, you must be familiar with the different memory types and the different data models available in the compiler. If you need to refresh these details, see the chapter *Data storage*.

STATIC MEMORY SEGMENTS

Static memory is memory that contains variables that are global or declared static, as described in 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, DATA16_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 data16 and __data16. The following table summarizes the memory types and the corresponding segment groups:

Memory type	Segment group	Memory range
Data16	DATA16	0x0-0x7FFF and 0xFFFF8000-0xFFFFFFFF
Data24	DATA24	0x0-0x7FFFFFFF and 0xFF800000-0xFFFFFFFF
Data32	DATA32	0x0-0xFFFFFFFF
Sbdata16	SBDATA16	SB + (0x0-0xFFFF)
Sbdata24	SBDATA24	SB + (0x0-0xFFFFF)

Table 10: Memory types with corresponding segment groups

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 details about segment memory types, see *Segment memory type*, page 28.

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
Non-initialized absolute addressed data	AN	
Constant absolute addressed data	AC	

Table 11: Segment name suffixes

For a list of all supported segments, see *Summary of segments*, page 225.

Examples

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

<pre>__data16 int j; __data16 int i = 0;</pre>	<p>The data16 variables that are to be initialized to zero when the system starts are placed in the segment DATA16_Z.</p>
<pre>__no_init __data16 int j;</pre>	<p>The data16 non-initialized variables are placed in the segment DATA16_N.</p>
<pre>__data16 int j = 4;</pre>	<p>The data16 non-zero initialized variables are placed in the segment DATA16_I in RAM, and the corresponding initializer data in the segment DATA16_ID in ROM.</p>

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 will fail:

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

However, in this 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 command file

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

```
//First, the segments to be placed in ROM are defined:
-Z (CONST) DATA16_C=FFFF8000-_USER_ROM_END
-Z (CONST) DATA24_C=_USER_ROM_BEGIN-_USER_ROM_END
-Z (CONST) DATA32_C
-Z (CONST) DATA16_ID, DATA24_ID, DATA32_ID

//Then, the RAM data segments are placed in memory:
-Z (DATA) DATA16_N, DATA16_Z, DATA16_I=_USER_RAM_BEGIN-04DF,
    0500-_USER_RAM_END
-Z (DATA) DATA24_N, DATA24_Z, DATA24_I
-Z (DATA) DATA32_N, DATA32_Z, DATA32_I
```

A line without a range assignment inherits the range from the previous line. All the data segments are placed in the area used by on-chip RAM. For information about the memory address symbols, see *Symbols for available memory areas*, page 31.

THE INTERNAL DATA STACK

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

The data segment used for holding the stack is called `CSTACK`. The system startup code initializes the stack pointer 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 size in the **Stack size** text box.



Stack size allocation from the command line

The size of the `CSTACK` segment is defined in the linker command file.

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

```
-D_CSTACK_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 without 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#_USER_RAM_BEGIN-_USER_RAM_END
```

Note:

- This range does not specify the size of the stack; it specifies the range of the available memory
- The `#` allocates the `CSTACK` segment at the end of the memory area. In practice, this means that the stack will get all remaining memory at the same time as it is guaranteed that it will be at least `_CSTACK_SIZE` bytes in size.



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. Because the second alternative is easier to detect, you should consider placing your stack so that it grows toward the end of the memory, if possible.

THE INTERRUPT STACK

In addition to the application stack segment, `CSTACK`, there is also a stack segment for some interrupts, `ISTACK`, that uses the `ISP` stack pointer register. In other respects, the information about `CSTACK` applies to `ISTACK` as well.

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:

- Linker segments used for the heap
- Allocating the heap size, which differs depending on which build interface you are using
- Placing the heap segments in memory.

The memory allocated to the heap is placed in the segment `HEAP`, which 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 size in the **Heap size** text box.



Heap size allocation from the command line

The size of the heap segment is defined in the linker command file.

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

```
-D_HEAP_SIZE=size
```

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

Specify the appropriate size for your application.



Placement of heap segment

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

```
-Z (DATA) HEAP+_HEAP_SIZE=_USER_RAM_BEGIN-04DF,0500-_USER_RAM_END
```

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



Heap size and standard I/O

If you excluded `FILE` descriptors from the DLIB runtime environment, as in the Normal configuration, there are no input and output buffers at all. Otherwise, as in the Full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an R32C/100

microcomputer. 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 `@` syntax, is placed in either the `DATA16_AC` or the `DATA16_AN` 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 command file.

If you create your own segments, these must also be defined in the linker command 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 a complete list of all segments, see *Summary of segments*, page 225. For information about the memory address symbols, see *Symbols for available memory areas*, page 31.

STARTUP CODE

The segment `CSTART` contains code used during system setup and runtime initialization (`cstartup`), and system termination (`cexit`). The segment must be placed into one continuous memory space, which means that the `-P` segment directive cannot be used.

NORMAL CODE

Functions declared without a memory type attribute are placed in different segments, depending on which code model you are using.

If you use the Far code model, or if the function is explicitly declared `__code24`, the code is placed in `CODE24` segment. If you use the Huge code model, or if the function is explicitly declared `__code32`, the code is placed in `CODE32` segment. Again, this is a simple operation in the linker command file:

```
-P (CODE) CODE24=_USER_ROM_BEGIN-_USER_ROM_END
-P (CODE) CODE32=_USER_ROM_BEGIN-_USER_ROM_END
```

The `-P` linker directive is used because it allows XLINK to split up the segments and pack their contents more efficiently. This is useful here, because the memory range is non-consecutive.

INTERRUPT VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The tables are placed in the segments `INTVEC` (normal interrupts) and `NMIVEC` (non-maskable interrupts and the reset vector). You must place the `NMIVEC` segment at a specific address, see *Interrupt functions*, page 23. The linker directives would then look like this:

```
-Z (CONST) NMIVEC=FFFFFFDC-__USER_ROM_END
-Z (CONST) INTVEC=__USER_ROM_BEGIN-__USER_ROM_END
```

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 (CODE) DIFUNCT=__USER_ROM_BEGIN-__USER_ROM_END
```

For additional information, see *DIFUNCT*, page 234.

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 command 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 further 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 **Range checks disabled** in the IDE, or the option `-R` on the command line, to generate an output file even if a range error was encountered.

For further information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IAR Embedded Workbench® IDE User Guide*.

The DLIB runtime environment

This chapter describes the runtime environment in which an application executes. In particular, the chapter covers the DLIB runtime library and how you can modify it—setting options, overriding default library modules, or building your own library—to optimize it for your application.

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

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

Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code. The IAR DLIB runtime environment can be used as is together with the debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

The *runtime environment* supports ISO/ANSI C and C++ including the standard template library. The runtime environment consists of the *runtime library*, which contains the functions defined by these standards, and include files that define the library interface.

The runtime library is delivered both as prebuilt libraries and as source files, and you can find them in the product subdirectories `r32c\lib` and `r32c\src\lib`, respectively.

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

- Support for hardware features:
 - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
 - Peripheral unit registers and interrupt definitions in include files
 - 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.
- Special compiler support for some functions, for instance functions for floating-point arithmetics.

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

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

LIBRARY SELECTION

To configure the most code-efficient runtime environment, you must determine your application and hardware requirements. The more functionality you need, the larger your code will become.

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

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s53`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: normal or full.

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

Note: Your application project must be able to locate the library, include files, and the library configuration file.

SITUATIONS THAT REQUIRE LIBRARY BUILDING

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

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 50.

LIBRARY CONFIGURATIONS

It is possible to configure the level of support for, for example, locale, file descriptors, multibyte characters. The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The configuration file is used for tailoring a build of a runtime library, and tailoring the system header files used when compiling your application. The less functionality you need in the runtime environment, the smaller it is.

These DLIB library configurations are available:

Library configuration	Description
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in <code>printf</code> and <code>scanf</code> , and no hex floats 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 hex floats in <code>strtod</code> .

Table 12: Library configurations

You can also 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 further information, see *Building and using a customized library*, page 50.

The prebuilt libraries are based on the default configurations, see Table 14, *Prebuilt libraries*, page 45. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

DEBUG SUPPORT IN THE RUNTIME LIBRARY

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 IDE	Linker command line option	Description
Basic debugging	Debug information for C-SPY	-Fubrof	Debug support for C-SPY without any runtime support
Runtime debugging	With runtime control modules	-r	The same as -Fubrof, but also includes debugger support for handling program abort, exit, and assertions.
I/O debugging	With I/O emulation modules	-rt	The same as -r, but also includes debugger support for I/O handling, which means that stdin and stdout are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.

Table 13: Levels of debugging support in runtime libraries

If you build your application project with the XLINK options **With runtime control modules** or **With I/O emulation modules**, certain functions in the library are replaced by functions that communicate with the debugger. For further information, see *C-SPY runtime interface*, page 65.



To set linker options for debug support in the IDE, choose **Project>Options** and select the **Linker** category. On the **Output** page, select the appropriate **Format** option.

Using a prebuilt library

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

- Code model
- Floating-point implementation
- Size of the double floating-point type
- Library configuration—Normal or Full.

These prebuilt runtime libraries are available:

Library file	Code model	Floating-point implementation	Size of double	Library configuration
d1r32cfhfn.r53	Far	Hardware	32 bits	Normal
d1r32cfhff.r53	Far	Hardware	32 bits	Full
d1r32cfhfn.r53	Far	Hardware	64 bits	Normal
d1r32cfhdf.r53	Far	Hardware	64 bits	Full
d1r32cfsfn.r53	Far	Software	32 bits	Normal
d1r32cfsff.r53	Far	Software	32 bits	Full
d1r32cfsdn.r53	Far	Software	64 bits	Normal
d1r32cfsdf.r53	Far	Software	64 bits	Full
d1r32chhfn.r53	Huge	Hardware	32 bits	Normal
d1r32chhff.r53	Huge	Hardware	32 bits	Full
d1r32chhfn.r53	Huge	Hardware	64 bits	Normal
d1r32chhdf.r53	Huge	Hardware	64 bits	Full
d1r32chsfn.r53	Huge	Software	32 bits	Normal
d1r32chsff.r53	Huge	Software	32 bits	Full
d1r32chsdn.r53	Huge	Software	64 bits	Normal
d1r32chsdf.r53	Huge	Software	64 bits	Full

Table 14: Prebuilt libraries

The names of the libraries are constructed in this way:

`<type><target><code_model><float><double><library_config>.r53`

where

- `<type>` is `d1` for the IAR DLIB runtime environment
- `<target>` is `r32c`
- `<code_model>` is either `f` for Far or `h` for Huge
- `<float>` is either `h` for hardware floating-point or `s` for software-emulated floating-point
- `<double>` is either `f` for 32 bits or `d` for 64 bits
- `<library_config>` is either `n` for Normal or `f` for Full.

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



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



If you build your application from the command line, you must specify the items to get the required runtime library:

- Specify which library object file to use on the XLINK command line, for instance:
`dlr32cfhdn.r53`
- Specify the include paths for the compiler and assembler:
`-I r32c\inc\dlib`
- Specify the library configuration file for the compiler:
`--dlib_config C:\...\dlr32cfhdn.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 `r32c\lib\`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the compiler can be used as is. However, it is possible to customize parts of a library without rebuilding it. There are two different methods:

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

These items can be customized:

Items that can be customized	Described in
Formatters for <code>printf</code> and <code>scanf</code>	<i>Choosing formatters for <code>printf</code> and <code>scanf</code></i> , page 47
Startup and termination code	<i>System startup and termination</i> , page 52
Low-level input and output	<i>Standard streams for input and output</i> , page 56
File input and output	<i>File input and output</i> , page 59
Low-level environment functions	<i>Environment interaction</i> , page 62
Low-level signal functions	<i>Signal and raise</i> , page 63
Low-level time functions	<i>Time</i> , page 63
Size of heaps, stacks, and segments	<i>Placing code and data</i> , page 27

Table 15: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 49.

Choosing formatters for printf and scanf

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

Note: If you rebuild the library, it is possible to optimize these functions even further, see *Configuration symbols for printf and scanf*, page 58.

CHOOSING PRINTF FORMATTER

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

This table summarizes the capabilities of the different formatters:

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

Table 16: Formatters for printf

† Depends on the library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 58.



Specifying the print formatter in the IDE

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



Specifying printf formatter from the command line

To use any other formatter than the default (`_PrintfLarge`), add one of these lines in the linker command file you are using:

```
-e_PrintfFull=_Printf
-e_PrintfSmall=_Printf
-e_PrintfTiny=_Printf
```

CHOOSING SCANF FORMATTER

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

This table summarizes the capabilities of the different formatters:

Formatting capabilities	_ScanfFull	_ScanfLarge	_ScanfSmall
Basic specifiers c, d, i, o, p, s, u, X, x, and %	Yes	Yes	Yes
Multibyte support	†	†	†
Floating-point specifiers a, and A	Yes	No	No
Floating-point specifiers e, E, f, F, g, and G	Yes	No	No
Conversion specifier n	Yes	No	No
Scan set [and]	Yes	Yes	No
Assignment suppressing *	Yes	Yes	No
long long support	Yes	No	No

Table 17: Formatters for scanf

† Depends on the library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 58.



Specifying scanf formatter in the IDE

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



Specifying scanf formatter from the command line

To use any other variant than the default (`_ScanfLarge`), add one of these lines in the linker command file you are using:

```
-e_ScanfFull=_Scanf
-e_ScanfSmall=_Scanf
```

Overriding library modules

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and `cstartup`. This can be done without rebuilding the entire library. This section describes the procedure for including your version of the module in the application project build process. The library files that you can override with your own versions are located in the `r32c\src\lib` directory.

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



Overriding library modules using the IDE

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

- 1 Copy the appropriate `library_module.c` file to your project directory.
- 2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.
- 3 Add the customized file to your project.
- 4 Rebuild your project.



Overriding library modules from the command line

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

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

```
iccr32c library_module
```

This creates a replacement object module file named `library_module.r53`.

Note: The size of the double floating-point type, the floating-point implementation, and the include paths must be the same for `library_module` as for the rest of your code.

- 4 Add `library_module.r53` to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module dlr32cfhdn.r53
```

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

Run XLINK to rebuild your application.

This will use your version of `library_module.r53`, instead of the one in the library. For information about the XLINK options, see the *IAR Linker and Library Tools Reference Guide*.

Building and using a customized library

In some situations, see *Situations that require library building*, page 43, it is necessary to rebuild the library. 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.

Information about the build process is described in the *IAR Embedded Workbench® IDE User Guide*.

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.

SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has Full library configuration, see Table 12, *Library configurations*, page 43.



In the IDE, modify the generic options in the created library project to suit your application, see *Basic settings for project configuration*, page 5.

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 `d1r32clibraryname.h`, which sets up that specific library with full library configuration. For more information, see Table 15, *Customizable items*, page 46.

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 `d1r32clibraryname.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:

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.s53`, `cexit.s53`, and `low_level_init.c` located in the `r32c\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 55.

SYSTEM STARTUP

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

For the hardware initialization, it looks like this:

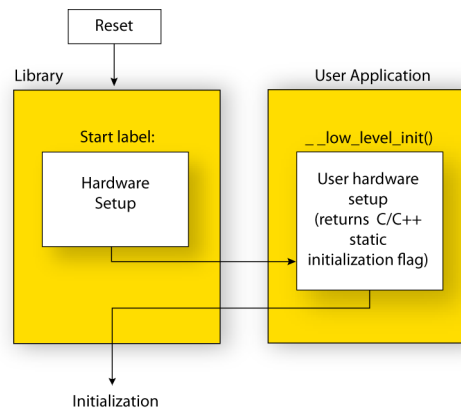


Figure 1: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code.
- Stack pointers, the `SB` register, and the interrupt vector base register are 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:

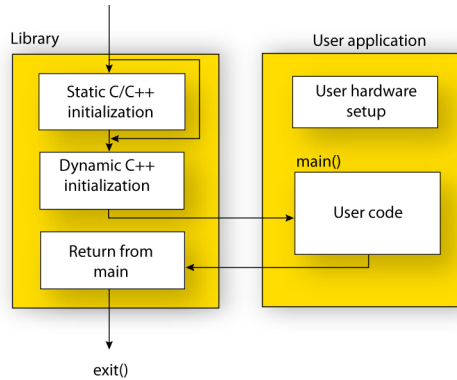


Figure 2: C/C++ initialization phase

- Static variables are initialized (if the return value of `__low_level_init` is non-zero). Zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. For more details, see *Initialized data*, page 33
- 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:

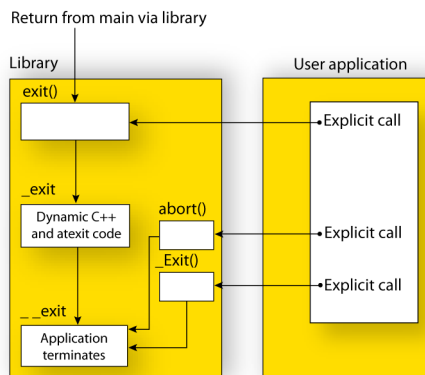


Figure 3: System termination phase

An application can terminate normally in two different ways:

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

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

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

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

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

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 *C-SPY runtime interface*, page 65.

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 providing a customized version of the routine `__low_level_init`, which is called from `cstartup.s53` before the data segments are initialized. Modifying the file `cstartup` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s53` and `low_level_init.c`, located in the `r32c\src\lib` directory.

Note: Normally, you do not need to customize the file `cexit.s53`.



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 50.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s53`, you do not have to rebuild the library.

__LOW_LEVEL_INIT

A skeleton low-level initialization file, `low_level_init.c`, is supplied with the product. 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 `low_level_init.c` to assure correct behavior of the `__low_level_init` routine.

MODIFYING THE FILE CSTARTUP.S53

As noted earlier, you should not modify the file `cstartup.s53` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify

the file `cstartup.s53`, 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 49.

Standard streams for input and output

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

There are primitive I/O functions, which are the fundamental functions through which C and C++ performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these functions using whatever facilities the hardware environment provides.

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `r32c\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 50. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY runtime interface*, page 65.

Example of using `__write` and `__read`

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

```
__no_init volatile unsigned char LCD_IO @ address;

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)
{
    LCD_IO = * Buf++;
    ++nChars;
}
return nChars;
}

```

Note: A call to `__write` where `BUF` has the value `NULL` is a command to flush the handle.

The code in this example uses memory-mapped I/O to read from a keyboard:

```

__no_init volatile unsigned char KB_IO @ address;

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 = KB_IO;
        if (c == 0)
            break;
        *Buf++ = c;
        ++nChars;
    }
    return nChars;
}

```

For information about the `@` operator, see *Controlling data and function placement in memory*, page 103.

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 47.

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
_DLIB_PRINTF_MULTIBYTE	Multibyte characters
_DLIB_PRINTF_LONG_LONG	Long long (ll qualifier)
_DLIB_PRINTF_SPECIFIER_FLOAT	Floating-point numbers
_DLIB_PRINTF_SPECIFIER_A	Hexadecimal floats
_DLIB_PRINTF_SPECIFIER_N	Output count (%n)
_DLIB_PRINTF_QUALIFIERS	Qualifiers h, l, L, v, t, and z
_DLIB_PRINTF_FLAGS	Flags -, +, #, and 0
_DLIB_PRINTF_WIDTH_AND_PRECISION	Width and precision
_DLIB_PRINTF_CHAR_BY_CHAR	Output char by char or buffered

Table 18: 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
_DLIB_SCANF_MULTIBYTE	Multibyte characters
_DLIB_SCANF_LONG_LONG	Long long (ll qualifier)
_DLIB_SCANF_SPECIFIER_FLOAT	Floating-point numbers
_DLIB_SCANF_SPECIFIER_N	Output count (%n)
_DLIB_SCANF_QUALIFIERS	Qualifiers h, j, l, t, z, and L
_DLIB_SCANF_SCANSET	Scanset ([*])
_DLIB_SCANF_WIDTH	Width
_DLIB_SCANF_ASSIGNMENT_SUPPRESSING	Assignment suppressing ([*])

Table 19: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must set up a library project, see *Building and using a customized library*, page 50. 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. If you use any of these functions, you must customize them to suit your hardware. To simplify adaptation to specific hardware, all I/O functions call a small set of primitive functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters.

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 43. 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 are 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 20: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 44.

Locale

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

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

The DLIB library can be used in two main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

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

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

The configuration symbol `_DLIB_FULL_LOCALE_SUPPORT`, which is defined in the library configuration file, determines whether a library has support for a locale interface

or not. The locale configuration symbols `_LOCALE_USE_LANG_REGION` and `_ENCODING_USE_ENCODING` define all the supported locales and encodings:

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C /* C locale */
#define _LOCALE_USE_EN_US /* US English */
#define _LOCALE_USE_EN_GB /* UK 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 50.

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 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
```

The last string must be empty. 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 `r32c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 49.

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 50.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 44.

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 `r32c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 49.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 50.

Time

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

This does not require that you rebuild the library. You can find source templates in the files `clock.c` and `time.c`, and `getzone.c` in the `r32c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 49.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 50.

The default implementation of `__getzone` specifies UTC as the time zone.

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY runtime interface*, page 65.

Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you must rebuild the library, see *Building and using a customized library*, page 50. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

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 `r32c\src\lib` directory. For further information, see *Building and using a customized library*, page 50. 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.

Hardware support

Some R32C/100 devices have a hardware floating-point unit (FPU). The R32C IAR C/C++ Compiler can take advantage of this and produce code which is very fast in floating-point operations.

FLOATING-POINT IMPLEMENTATION

The native floating-point operations of the FPU are not fully standards-compliant. For this reason, there are three different implementations of floating-point support. These are:

- Software-emulated floating-point operations
- Native FPU floating-point operations
- FPU floating-point operations with standards-compliant library functions

Software-emulated floating-point operations



If no FPU is available, you should specify the Full floating-point model, `--fp_model=full`. This will use the library routines that emulate floating-point operations.



In the IAR Embedded Workbench IDE, select **Project>Options>General Options>Target>Float implementation>Software emulation**.

Native FPU floating-point operations



If an FPU is available, you can specify `--fp_model=fast`. This will cause FPU instructions to be inserted directly in the code. This is the fastest model. However, in this model, the special numbers Not a number (NaN) and Infinity are not recognized.



In the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target** and select **FPU only**.

FPU with standards-compliant library functions



If your application uses Not a number (NaN) or Infinity, you can use `--fp_model=full` together with an extra linker command file `fpu_compliant.xcl` to use alternative floating-point library functions that recognize Infinity and Not a Number, but still use the FPU instructions to perform the operations. This creates a significant overhead compared to the inlined FPU instructions of `--fp_model=fast`, but it is approximately twice as fast as the software-emulating library.

Note: If you specify `--fp_model=full` without using the extra linker command file, software-emulated floating-point operations will be used.



In the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target** and select **Standards-compliant with FPU**.

See also *Floating-point types*, page 156.

C-SPY runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 44.

In this case, C-SPY variants of these library functions are linked to the application:

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

Table 21: Functions with special meanings when linked with debug info

Function	Description
time	Returns the time on the host computer
__write	stdin, stdout, and stderr will be directed to the Terminal I/O window, all other files will write to the associated host file

Table 21: Functions with special meanings when linked with debug info (Continued)

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

LOW-LEVEL DEBUGGER RUNTIME INTERFACE

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

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

The 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 options for C-SPY runtime interface. 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 DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Debug support in the runtime library*, page 44. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

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

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

Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the IAR compiler, assembler, and linker to ensure module consistency.

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the compiler, it is possible to specify the size of the `double` floating-point type. If you write a routine that only works for 64-bit doubles, it is possible to check that the routine is not used in an application built using 32-bit doubles.

The tools provided by IAR Systems use a set of predefined runtime model attributes. You can use these predefined attributes or define your own to perform any type of consistency check.

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

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

Table 22: Example of runtime model attributes

USING RUNTIME MODEL ATTRIBUTES

To ensure module consistency with other object files, use the `#pragma rtmodel` directive to specify runtime model attributes in your C/C++ source code. For example:

```
#pragma rtmodel="__rt_version", "1"
```

For detailed syntax information, see *rtmodel*, page 196.

You can also use the `RTMODEL` assembler directive to specify runtime model attributes in your assembler source code. For example:

```
RTMODEL "color", "red"
```

For detailed syntax information, see the *R32C IAR Assembler Reference Guide*.

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 Systems 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>__core</code>	R32C	Identifies the microcomputer core.

Table 23: Predefined runtime model attributes

Runtime model attribute	Value	Description
__double_size	32 or 64	The size, in bits, of the double floating-point type.
__rt_version	<i>n</i>	This runtime key is always present in all modules generated by the R32C IAR C/C++ Compiler. If a major change in the runtime characteristics occurs, the value of this key changes.

Table 23: 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, refer to the chapter *Assembler directives* in the *R32C IAR Assembler Reference Guide*.

Example

The following assembler source code provides a function that increases the register `R4` to count the number of times it was called. The routine assumes that the application does not use `R4` for anything else, that is, the register is locked for usage. To ensure this, a runtime module attribute, `__reg_r4`, 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. Note that the compiler sets this attribute to `free`, unless the register is locked.

```
RTMODEL      "__reg_r4", "counter"
MODULE       myCounter
PUBLIC       myCounter
RSEG        CODE:CODE:NOROOT(1)
myCounter:  INC      R4
            RET
            ENDMOD
            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_r4' must be 'counter', but module part1
has the value 'free'
```

USER-DEFINED RUNTIME MODEL ATTRIBUTES

In cases where the predefined runtime model attributes are not sufficient, you can use the `RTMODEL` assembler directive to define your own attributes. For each property, select

a key and a set of values that describe the states of the property that are incompatible. Note that key names that start with two underscores are reserved by the compiler.

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"
```

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 R32C/100 microcomputer 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 R32C provides several ways to mix C or C++ and assembler:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

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

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

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

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

The first issue is discussed in the section *Calling assembler routines from C*, page 74. The following two are covered in the section *Calling convention*, page 77.

The section on memory access methods, page 86, covers how data in memory is accessed.

The answer to the final question is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the *call frame*, which must be supplied as annotations in the assembler source file. For more information, see *Call frame information*, page 88.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 74, and *Calling assembler routines from C++*, page 76, respectively.

INLINE ASSEMBLER

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

```
bool flag;

void foo(void)
{
    while (!flag)
    {
        asm("MOV PIND, flag");
    }
}
```

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

The inline assembler instruction will simply be inserted at the given location in the program flow. The consequences or side-effects the insertion might have on the surrounding code are not taken into consideration. If, for example, registers or memory locations are altered, they might have to be restored within the sequence of inline assembler instructions for the rest of the code to work properly.

Inline assembler sequences have no well-defined interface with the surrounding code generated from your C or C++ code. This makes the inline assembler code fragile, and

will possibly also become a maintenance problem if you upgrade the compiler in the future. There are also several limitations to using inline assembler:

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Alignment cannot be controlled; this means, for example, that `DC32` directives might be misaligned
- Auto variables cannot be accessed.

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 `double`, and then returns an `int`:

```
extern int gInt;
extern double gDouble;

int func(int arg1, double arg2)
{
    int locInt = arg1;
    gInt = arg1;
    gDouble = arg2;
    return locInt;
}

int main()
{
    int locInt = gInt;
    gInt = func(locInt, gDouble);
    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:

```
iccr32c skeleton -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 `s53`. Also remember to specify:

- the code model
- the data model
- the floating point model
- the size of the `double` type

- a low level of optimization
- `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s53`.

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 compiler runtime 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.

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 my_routine(int x);
}
```


Memory access layout of non-PODs (“plain old data structures”) is not defined, and might change between compiler versions. Therefore, we do not recommend that you access non-PODs from assembler routines.

To achieve the equivalent to a non-static member function, the implicit `this` pointer must be made explicit:

```
class X;

extern "C"
{
    void doit(X *ptr, int arg);
}
```

It is possible to “wrap” the call to the assembler routine in a member function. Using an inline member function removes the overhead of the extra call—provided that function inlining is enabled:

```
class X
{
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 a_function(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++:

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

    int f(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general R32C/100 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.

The registers R2R0 and A0, as well any registers used for parameters (for instance R3R1, R7R5, A1, and A2), can be used as scratch registers 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 R6R4 and A3 are preserved registers, and all non-parameter registers except for R2R0 and A0.

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 frame pointer register FB must be restored before returning from the function.
- The SB register (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.

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

- 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, the memory location where the structure will be stored is passed in the register `A0` as a hidden parameter.

Register parameters

The registers available for passing parameters are `R0–R3`, `R5`, `R7`, and `A0–A2`.

Parameters	Passed in registers
8-bit values	R0L, R0H, R2L, R2H, R1L, R1H, R3L, R3H
16-bit values	R0, R2, R1, R3, R5, R7
32-bit values	R2R0, R3R1, R7R5, A0, A1, A2
32-bit values (pointer)	A0, A1, A2
64-bit values	R3R1R2R0, A1A0

Table 24: Registers used for passing parameters

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 suitable register of the appropriate size available, the parameter is passed on the stack.

The registers are selected in the order given in Table 24, *Registers used for passing parameters*.

For example, the two functions `g` and `h` both take two `int`'s, four `char`'s and a `short` parameter.

```
void g(int,int,char,char,char,char,short);
void h(char,char,char,char,short,int,int);
```

When the first two parameters of the function `g` have been allocated (to `R2R0` and `R3R1` respectively), there are no remaining 8-bit registers for passing the `char` parameters, which will be passed on the stack. Register `R5` is still free though, so the `short` parameter will be passed in `R5`.

The function `h` on the other hand, will pass the `char` parameters in registers `R0L`, `R0H`, `R2L`, and `R2H`. Because there are many parameter registers left, all parameters can be

passed in registers. The `short` parameter goes into R1, and the two `int` parameters in R7R5 and A0.

Both functions have the same types and number of parameters, but `h` will be significantly more efficient because all parameters are passed in registers. The function `g` will also use 16 bytes of stack space for passing the `char` parameters, whereas function `h` does not need stack parameters.

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 four, etc.

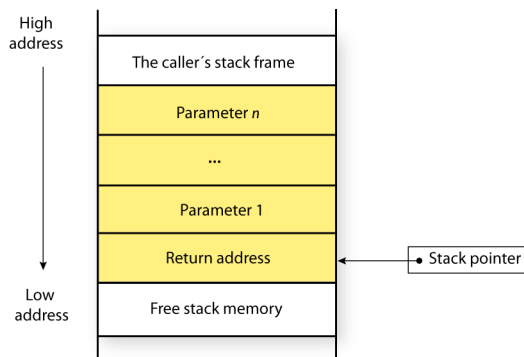


Figure 4: Stack image after the function call

And this is what the stack looks like when the execution has reached the first statement in the function:

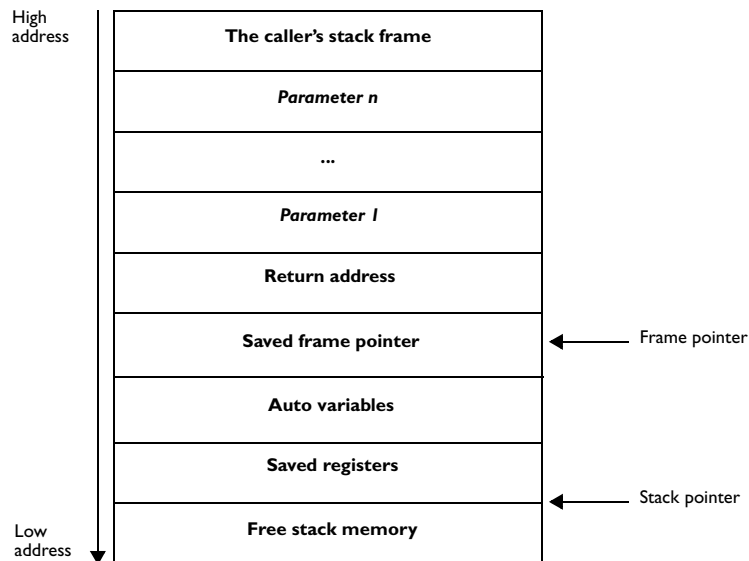


Figure 5: Stack image on the first statement of the function, after the function call

Aligning the function entry point

The runtime performance of a function depends on the entry address assigned by the linker. To make the function execution time less dependent on the entry address, the alignment of the function entry point can be specified explicitly using a compiler option, see `--align_func`, page 130. A higher alignment does not necessarily make the function faster, but the execution time will be more predictable.

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

These are the registers available for returning values:

Return values	Returned in registers
8-bit values	R0L
16-bit values	R0
32-bit values	R2R0 (scalar) A0 (pointer)
64-bit values	R3R1R2R0
struct values	implicit pointer in A0

Table 25: Registers used for returning values

Stack layout at function exit

It is the responsibility of the caller to clean the stack after the called function returns, by adjusting the stack pointer.

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 on the stack.

Typically, a function returns by using the `RTS` instruction or the `EXITD` instruction if the `FB` register is used for stack accesses. Interrupt functions return by using the `REIT` or `EXITI` instructions, respectively, and fast interrupt functions (declared using the `__fast_interrupt` keyword) return by using the `FREIT` instruction.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

Functions declared `__monitor` save the status register together with the saved registers. Functions declared `__interrupt` save all used registers. Functions declared `__task` do not save any registers.

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 R2R0, and the return value is passed back to its caller in the register R2R0.

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

```
ADD.L    #1, R2R0
RTS
```

Example 2

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

```
struct a_struct { int a; };
int a_function(struct a_struct x, int y);
```

The calling function must reserve 4 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 R2R0. The return value is passed back to its caller in the register R2R0.

Example 3

The function below will return a `struct`.

```
struct a_struct { int a; };
struct a_struct a_function(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 A0. The parameter `x` is passed in R2R0.

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

```
struct a_struct * a_function(int x);
```

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in R2R0, and the return value is returned in A0.

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 R32C 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 reference information about the function directives, see the *R32C IAR Assembler Reference Guide*.

Calling functions

In this section, we describe how functions are called in the different code models.

Functions can be called in two fundamentally different ways—directly or via a function pointer. For each code model, we will discuss how both types of calls will be performed.

ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

This section presents the assembler instructions that can be used for calling and returning from functions on the R32C/100 microcomputer. In the following sections we will see how the different code models use these instructions to call code.

The normal function calling instruction is the JSR instruction:

```
JSR.A    label:24
```

The location that the called function should return to (that is, the location immediately after this instruction) is stored on the stack.

The destination label must not be further away than 8 Mbytes. Larger jumps can be made with the JSRI instruction.

```
MOV.L    #label:32,A0
JSRI.L   A0
```

Far code model

A direct call using this code model is simply:

```
JSR.A    function:24
```

The JSR instruction can only reach ± 8 Mbytes.

When a function returns control to the caller, the RTS instruction is used if no auto variables were pushed at function entry and the EXITD instruction is used if auto variables were saved at function entry.

When a function call is made via a function pointer, this code will be generated:

```
JSRI.L   function_pointer:24    ; Location of function pointer
```

Calls via a function pointer reach the whole 32-bit address space.

Huge code model

In the Huge code model, a function generates an indirect call. This example shows a simple call to the function `func`:

```
MOV.L    #func:32,A0
JSRI.L   A0
```

Returning from a function and assigning a function pointer work as in the Far code model.

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, it will be used for explaining the reason behind the different memory types.

You should be familiar with the R32C/100 instruction set, in particular the different addressing modes used by the instructions that can access memory.

The IAR Assembler for R32C uses the convention that the size of a memory address is controlled by use of the suffixes `:8`, `:16`, and `:24`. For example:

Suffix	Size	Instruction example
:8	1 byte	MOV.L -8:8[FB],R2R0
:16	2 bytes	MOV.L foo:16[FB],R2R0
:24	3 bytes	MOV.L bar:24,R2R0

Table 26: Specifying the size of a memory address in assembler instructions

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 x;
char y[10];

struct s
{
    long a;
    char b;
};
```

```
char test(int i, struct s * p)
{
    return x + y[i] + p->b;
}
```

THE DATA16 MEMORY ACCESS METHOD

Data16 memory is located in the lowest and highest 32 Kbytes of memory. This is the only memory type that can be accessed using 16-bit addresses.

Examples

These examples access data16 memory in different ways:

MOV.B:S	x:16,R0L	Access the global variable x
ADD.B	y:16[A1],R0L	Access an entry in the global array y
ADD.B	0x4:8[A0],R0L	Access through a pointer

THE DATA24 MEMORY ACCESS METHOD

The first 8 Mbytes of memory can be accessed using 3-byte addresses.

Examples

These examples access data24 memory in different ways:

MOV.B	x:24,R0L	Access the global variable x
ADD.B	y:24[A1],R0L	Access an entry in the global array y
ADD.B	0x4:8[A0],R0L	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 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.

Examples

These examples access data32 memory in different ways:

MOV.L	#x:32,A1	Access the global variable x
MOV.B	[A1],R0L	
MOV.L	#y:32,A1	Access an entry in the global array y
INDEX1.L	R3R1	
ADD.B	[A1],R0L	

```
ADD.B      0x4:8[A0],R0L      Access through a pointer
```

THE SBDATA16 MEMORY ACCESS METHOD

Sbdata16 memory is addressed relative to the `SB` register. An address in this memory is a 2-byte unsigned offset to the base address in the `SB` register. The start of the Sbdata area is marked by the `SBREF` symbol, defined in the linker command file.

Examples

This example accesses sbdata16 memory:

```
MOV.L      x-SBREF:16[SB],R2R0  Accesses the __sbdata16 variable x
```

THE SBDATA24 MEMORY ACCESS METHOD

Sbdata24 memory is addressed relative to the `SB` register. An address in this memory is a 3-byte unsigned offset to the base address in the `SB` register. The start of the Sbdata area is marked by the `SBREF` symbol, defined in the linker command file.

Examples

This example accesses sbdata24 memory:

```
MOV.L      y-SBREF:24[SB],A0    Accesses the __sbdata24 variable y
```

Call frame information

When debugging an application using C-SPY, it is possible to 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 *R32C IAR Assembler Reference Guide*.

The CFI directives will 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	The call frame of the stack
R4–R7, R0L–R3L, R0H–R3H	General purpose data registers
A0–A3	General purpose address registers
?RET32	The return address
SB	The static base register
FB	The frame base register
FLG	The flag register
SP	The active stack pointer (USB or ISB depending on the FLG register U bit)
ISP	The interrupt stack pointer

Table 27: Call frame information resources defined in a names block

Example

The header file `cfi.m53` 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.

This is an example of an assembler routine that stores a permanent register and the return register to the stack:

```
#include "cfi.m53"

XCFI_NAMES    myNames
XCFI_COMMON   myCommon, myNames
MODULE        cfexample

PUBLIC        cfexample

RSEG          CODE24:CODE:NOROOT
```

```

CFI          Block myBlock Using myCommon
CFI          Function 'cfiexample'

// The common block does not declare the scratch
// registers as undefined.
CFI          R0L Undefined
              R0H Undefined
              R2L Undefined
              R2H Undefined

cfiexample:
    PUSHM     R3R1

    CFI       ?RET32 Frame(CFA_SP, -4)
    CFI       R3R1 Frame(CFA_SP, -8)
    CFI       CFA_SP SP+8

    // Do something useless just to demonstrate the
    // call stack.

    MOV.L     #0, R3R1
    MOV.L     #0, R2R0

    POPM      R3R1

    CFI       ?RET32 RA
    CFI       R3R1 SameValue
    CFI       CFA_SP SP

    // Do something else.

    MOV.L     #0, R2R0

    RTS

    CFI       ENDBLOCK myBlock
    ENDMOD

    END

```

Using C++

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

Overview

Embedded C++ is a subset of the C++ programming language which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language.

STANDARD EMBEDDED C++

The following C++ features are supported:

- Classes, which are user-defined types that incorporate both data structure and behavior; the essential feature of inheritance allows data structure and behavior to be shared among classes
- Polymorphism, which means that an operation can behave differently on different classes, is provided by virtual functions
- Overloading of operators and function names, which allows several operators or functions with the same name, provided that their argument lists are sufficiently different
- Type-safe memory management using the operators `new` and `delete`
- Inline functions, which are indicated as particularly suitable for inline expansion.

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are recent additions to the ISO/ANSI C++ standard. This is because they represent potential portability problems, due to that few development tools support the standard. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

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

- Templates
- Multiple and virtual inheritance
- Exception handling

- 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 C++ SUPPORT



In the compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 137.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 138.



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

Feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for R32C, you must be aware of some benefits and some possible quirks when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

CLASSES

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

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

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

For further information about attributes, see *Type qualifiers*, page 161.

Example

```
class A {
public:
    static __data16 __no_init int i @ 60; //Located in data16 at
                                         //address 60
    static __code24 void f(); //Located in code24 memory
    __code24 void g();        //Located in code24 memory
    virtual __code24 void h(); //Located in code24 memory
};
virtual void m() const volatile @ "SPECIAL"; //m() placed in
                                             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.

Example

```
class B {
public:
    void f();
    int i;
};
```

FUNCTIONS

A function with `extern "C"` linkage is compatible with a function that has C++ linkage.

Example

```
extern "C" {
    typedef void (*fpC)(void); // A C function typedef
}
void (*fpCpp)(void);          // A C++ function typedef

fpC f1;
fpCpp f2;
void f(fpC);

f(f1);                        // Always works
f(f2);                        // fpCpp is compatible with fpC
```

TEMPLATES

Extended EC++ supports templates according to the C++ standard, except for the support of the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in *Extended Embedded C++*, page 92.

STL and the IAR C-SPY® Debugger



C-SPY has built-in display support for the STL containers. The logical structure of containers is presented in the watch views in a comprehensive way that is easy to understand and follow.

Note: To be able to watch STL containers with many elements in a comprehensive way, the **STL container expansion** option—available by choosing **Tools>Options>Debugger**—is set to display only a few items at first.

VARIANTS OF CASTS

In Extended EC++ these additional C++ cast variants can be used:

```
const_cast<t2>(t)
static_cast<t2>(t)
reinterpret_cast<t2>(t)
```

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 // Nothing here
```

USING INTERRUPTS AND EC++ DESTRUCTORS

If interrupts are enabled and the interrupt functions use class objects that have destructors, there might be problems if the program exits either by using `exit` or by returning from `main`. If an interrupt occurs after an object was destroyed, there is no guarantee that the program will work properly.

To avoid this, make sure that interrupts are disabled when returning from `main` or when calling `exit` or `abort`.

To avoid interrupts, place a call to the intrinsic function `__disable_interrupt` before the call to `_exit`.

C++ language extensions

When you use the compiler in C++ mode and enable IAR language extensions, the following C++ language extensions are available in the compiler:

- In a friend declaration of a class, the `class` keyword can be omitted, for example:

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

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

```
class A {
    const int size = 10; //Possible when using IAR language
                        //extensions
    int a[size];
};
```

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 f();    //According to standard
};
```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

```
extern "C" void f(); //Function with C linkage
void (*pf) ()        //pf points to a function with C++ linkage
    = &f; //Implicit conversion of 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:

```
char *P = x ? "abc" : "def"; //Possible when using IAR
                             //language extensions
char const *P = x ? "abc" : "def"; //According to standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.

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

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
- Writing efficient code.

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 unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values. This applies especially to loop variables.
- Try to avoid 64-bit data types, such as 64-bit `double` and `long long`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

For details about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a floating-point unit is very inefficient, both in terms of code size and execution speed. 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.

For target devices equipped with a floating-point unit, you have two options with regard to 32-bit floating-point numbers: using the FPU's own floating-point instructions, which do not comply fully with the IEEE-754 standard, or using library functions that are standards-compliant but slower. See *Floating-point implementation*, page 64.

If available, use the FPU's own floating-point instructions unless you absolutely cannot manage without support for Infinity and NaN.

In the IAR C/C++ Compiler for R32C, the floating-point type `float` always uses the 32-bit format. The format used by the `double` floating-point type depends on the setting of the **Size of type 'double'** compiler option.

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

Note that 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`, 1 is added and the result is converted back to a `float`:

```
float test(float a)
{
    return a + 1.0;
}
```

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

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


CASTING A FLOATING-POINT VALUE TO AN INTEGER

If you want the result of casting a `float` to an `int` to be a rounded value instead of a truncated value, use the intrinsic function `__ROUND` to insert a `ROUND` instruction directly into the code. See `__ROUND`, page 206.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The R32C/100 microcomputer requires that data in memory must be aligned. Each element in a structure must be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment correct.

There are two reasons why this can be considered a problem:

- Due to external demands; for example, network communication protocols are usually specified in terms of data types with no padding in between
- You need to save data memory.

For information about alignment requirements, see *Alignment*, page 153.

There are two ways to solve the problem:

- Use the `#pragma pack` directive or the `__packed` data type attribute for a tighter layout of the structure. The drawback is that each access to an unaligned element in the structure becomes slower.
- Write your own customized functions for *packing* and *unpacking* structures. This is a more portable way, which will not produce any more code apart from your functions. The drawback is the need for two views on the structure data—packed and unpacked.

For further details about the `#pragma pack` directive, see *pack*, page 194.

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 R32C 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 137, 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 tag;
    union
    {
        long l;
        float f;
    };
} st;

void f(void)
{
    st.l = 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;
    };
} @ address;
```

This declares an I/O register byte `IOPORT` at `address`. The I/O register has 2 bits declared, `way` and `out`. Note that both the inner structure and the outer union are anonymous.

This example illustrates how variables declared this way can be used:

```
void test(void)
{
    IOPORT = 0;
    way = 1;
    out = 1;
}
```

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 microcomputer and thereby also place functions and data objects in different parts of memory. To read more about data and code models, see *Data models*, page 12, and *Code models and memory attributes for function storage*, page 21, respectively.

- Memory attributes

Use memory attributes to override the default addressing mode and placement of individual functions and data objects. To read more about memory attributes for data and functions, see *Using data memory attributes*, page 14, and *Using function memory attributes*, page 22, respectively.

- 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, for example variables with external requirements, or for populating any hardware tables similar to interrupt vector 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.

At compile time, data and functions are placed in different segments as described in *Data segments*, page 31, and *Code segments*, page 37, 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 command file, as described in *Placing segments in memory*, page 28.

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.

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

Examples

In this example, a `__no_init` declared variable is placed at an absolute address. This is useful for interfacing between multiple processes, applications, etc:

```
__no_init volatile char alpha @ 0x2000; /* OK */
```

These examples contain 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=0x60000
__no_init const int beta;                /* OK */

const int gamma @ 0x60004 = 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".*/
```

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:

```
extern volatile const __no_init int x @ 0x100;  /* the extern
                                                /* keyword makes x public */
```

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 `@` 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.

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 command 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 command file.

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

Examples of placing variables in named segments

In the following three examples, a data object is placed in a user-defined segment. The segment will be allocated in default memory depending on the used data model.

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

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

const int gamma @ "CONSTANTS" = 3; /* OK */
```

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__data32 __no_init int alpha @ "NOINIT"; /* Placed in data32*/
```

This example shows incorrect usage:

```
int delta @ "NOINIT";              /* Error, neither */
                                   /* "__no_init" nor "const" */
```

Examples of placing functions in named segments

```
void f(void) @ "FUNCTIONS";

void g(void) @ "FUNCTIONS"
{
}

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

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__code32 void f(void) @ "FUNCTIONS";
```

Controlling compiler optimizations

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

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

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to *optimize*, page 193, 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 141.

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 136.

OPTIMIZATION LEVELS

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

Optimization level	Description
None (Best debug support)	Variables live through their entire scope

Table 28: Compiler optimization levels

Optimization level	Description
Low	Dead code elimination
	Redundant label elimination
	Redundant branch elimination
Medium	Same as above
	Live-dead analysis and optimization
	Code hoisting
	Register content analysis and optimization
	Constant propagation
	Constant elimination
High (Balanced)	Common subexpression elimination
	Same as above
	Peephole optimization
	Cross jumping
	Memory contents tracking
	Copy propagation
	Cross call (when optimizing for size)
	Loop unrolling
	Function inlining
	Code motion
	Type-based alias analysis

Table 28: Compiler optimization levels (Continued)

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 109.

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

SPEED VERSUS SIZE

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

FINE-TUNING ENABLED TRANSFORMATIONS

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

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

To read more about the command line option, see `--no_cse`, page 143.

Loop unrolling

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

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

To read more about the command line option, see `--no_unroll`, page 146.

Function inlining

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

The compiler decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

To read more about the command line option, see `--no_inline`, page 144.

Code motion

Evaluation of loop-invariant expressions and common subexpressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization level **High**, normally reduces code size and execution time. The resulting code might however be difficult to debug.

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

Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object are performed using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers can reference the same memory location or not.

Type-based alias analysis is performed at optimization level **High**. For ISO/ANSI standard-conforming C or C++ application code, this optimization can reduce code size and execution time. However, non-standard-conforming C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

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

To read more about the command line option, see `--no_tbaa`, page 145.

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. This optimization cannot be disabled using the `#pragma optimize` directive.

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

To read more about related command line options, see `--no_cross_call`, page 143.

Writing efficient code

This section contains general programming hints on how to implement functions to make your applications robust, but at the same time facilitate compiler optimizations.

The following is a list of programming techniques that will, when followed, enable the compiler to better optimize the application.

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining might enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive and the C++ keyword `inline` gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the `--no_inline` command line option; see `--no_inline`, page 144.

- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more details, see *Mixing C and assembler*, page 71.

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
- Pass smaller parameters before larger ones, so that you avoid passing parameters on the stack. See *Register parameters*, page 80.

FUNCTION PROTOTYPES

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

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

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

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

Prototyped style

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

```
int test(char, int);           /* declaration */
int test(char a, int b)       /* definition */
{
    .....
}
```

Kernighan & Ritchie style

In K&R style—traditional pre-ISO/ANSI C—it is not possible to declare a function prototyped. Instead, an empty parameter list is used in the function declaration. Also, the definition looks different.

```

int test();                               /* old declaration */
int test(a,b)                             /* old definition */
char a;
int b;
{
    .....
}

```

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 0xFFFFF7F. On the left hand side, c1 is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 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. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 161.

A sequence that accesses a `volatile` declared variable must also not be interrupted. Use the `__monitor` keyword in interruptible code to ensure this. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. This is true for all variables of all sizes. Accessing a small-sized variable can be an atomic operation, but this is not guaranteed and 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.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several R32C/100 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. The following example is from `ior32c121.h`:

```
/* uart0 transmit receive mode Register */
__data16 __no_init volatile union
{
    unsigned char U0MR;
    struct
    {
        unsigned char SMD0      : 1;
        unsigned char SMD1      : 1;
        unsigned char SMD2      : 1;
        unsigned char CKDIR      : 1;
        unsigned char STPS      : 1;
        unsigned char PRY        : 1;
        unsigned char PRYE       : 1;
        unsigned char IOPOL      : 1;
    } U0MR_bit;
} @ 0x0368;
```

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:

```
/* Whole register access */
U0MR = 0x12;

/* Bitfield accesses */
U0MR_bit.CKDIR = 0;
U0MR_bit.IOPOL = 1;
```

You can also use the header files as templates when you create new header files for other R32C/100 devices. For details about the @ operator, see *Located data*, page 37.

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.

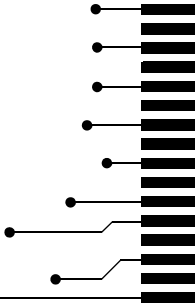
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For information about the `__no_init` keyword, see page 181. Note that to use this keyword, language extensions must be enabled; see *-e*, page 137. For information about the `#pragma object_attribute`, see page 193.

Part 2. Reference information

This part of the IAR C/C++ Compiler for R32C Reference Guide contains these chapters:

- External interface details
- Compiler options
- Data representation
- Compiler extensions
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- Segment reference
- Implementation-defined behavior.





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. Refer to the *IAR Embedded Workbench® IDE User Guide* for information about using the compiler from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccr32c [options] [sourcefile] [options]
```

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

```
iccr32c prog --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order that they are specified on the command line.

If you run the compiler from the command line without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

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 `iccr32c` command, either before or after the source filename; see *Invocation syntax*, page 119.

- Via environment variables
The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 120.
- Via a text file, using the `-f` option; see `-f`, page 139.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *Compiler options* chapter.

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: C_INCLUDE=c:\program files\iar systems\embedded workbench 5.n\r32c\inc;c:\headers
QCCR32C	Specifies command line options; for example: QCCR32C=-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, 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 140.
 - 2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 120.
 - 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:

```
iccr32c ..\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.

Note: Both `\` and `/` can be used as directory delimiters.

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 `.o`.
- Optional list files
Different types of list files can be specified using the compiler option `-l`, see `-l`, page 140. 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 `stderr` and displayed on the screen, and printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 122.
- 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 122.
- Size information
Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` 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,linenumber level[tag]: message
```

with these elements:

<i>filename</i>	The name of the source file in which the issue was encountered
<i>linenumber</i>	The line number at which the compiler detected the issue
<i>level</i>	The level of seriousness of the issue
<i>tag</i>	A unique tag that identifies the diagnostic message
<i>message</i>	An explanation, possibly several lines long

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

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

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 construction that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 150.

Warning

A diagnostic message that is produced when the compiler finds a programming error or omission which is of concern, but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see page 146.

Error

A diagnostic message that is produced when the compiler finds a construction 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 128, for a description of the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler. 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.



Refer to the *IAR Embedded Workbench® IDE User Guide* 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 119.

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=filename
```

or

```
--diagnostics_tables filename
```

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 filename
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n filename
```

Rules for specifying a filename or directory as parameters

These rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally also take a path. The path can be relative or absolute. For example, to generate a listing to the file `list.lst` in the directory `..\listings\`:

```
iccr32c prog -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:

```
iccr32c prog -l ../listings\
```

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

- The *current directory* is specified with a period (`.`). For example:
- ```
iccr32c prog -l .
```
- `/` can be used instead of `\` as the directory delimiter.
  - By specifying `-`, input files and output files can be redirected to `stdin` and `stdout`, respectively. For example:

```
iccr32c prog -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`:

```
iccr32c prog -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                                                     |
|--------------------------|-----------------------------------------------------------------|
| --align_func             | Specifies the alignment of the function entry point             |
| --char_is_signed         | Treats char as signed                                           |
| --code_model             | Specifies the code model                                        |
| -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_config            | Determines the library configuration file                       |
| --double                 | Sets the size of the data type double                           |
| -e                       | Enables language extensions                                     |
| --ec++                   | Enables Embedded C++ syntax                                     |
| --eec++                  | Enables Extended Embedded C++ syntax                            |
| --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                                        |
| --fp_model               | Specifies the floating-point implementation                     |
| --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                                        |
| --mfc                    | Enables multi file compilation                                  |

Table 31: Compiler options summary

| Command line option                              | Description                                                                                                                                                                                         |
|--------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--migration_preprocessor_extensions</code> | Extends the preprocessor                                                                                                                                                                            |
| <code>--misrac</code>                            | Enables error messages specific to MISRA-C:1998. This option is a synonym of <code>--misrac1998</code> and is only available for backwards compatibility.                                           |
| <code>--misrac1998</code>                        | Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .                                                                              |
| <code>--misrac2004</code>                        | Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .                                                                              |
| <code>--misrac_verbose</code>                    | Enables verbose logging of MISRA C checking. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> or the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> , respectively. |
| <code>--module_name</code>                       | Sets the object module name                                                                                                                                                                         |
| <code>--no_code_motion</code>                    | Disables code motion optimization                                                                                                                                                                   |
| <code>--no_cross_call</code>                     | Disables cross-call 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_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                                                                                                                                                                            |
| <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.                                                                                                                                                                       |

Table 31: Compiler options summary (Continued)

| Command line option         | Description                                                  |
|-----------------------------|--------------------------------------------------------------|
| --preinclude                | Includes an include file before reading the source file      |
| --preprocess                | Generates preprocessor output                                |
| --public_equ                | Defines a global named assembler label                       |
| -r                          | Generates debug information                                  |
| --remarks                   | Enables remarks                                              |
| --require_prototypes        | Verifies that functions are declared before they are defined |
| --silent                    | Sets silent operation                                        |
| --strict_ansi               | Checks for strict compliance with ISO/ANSI C                 |
| --warnings_affect_exit_code | Warnings affects exit code                                   |
| --warnings_are_errors       | Warnings are treated as errors                               |

Table 31: Compiler options summary (Continued)

## Descriptions of options

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



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

### --align\_func

Syntax

--align\_func={1|2|4|8}

Parameters

- |             |                                                            |
|-------------|------------------------------------------------------------|
| 1 (default) | Sets the alignment of the function entry point to 1 byte.  |
| 2           | Sets the alignment of the function entry point to 2 bytes. |
| 4           | Sets the alignment of the function entry point to 4 bytes. |
| 8           | Sets the alignment of the function entry point to 8 bytes. |

Description

Use this option to specify the alignment of the function entry point.

See also

*Aligning the function entry point*, page 82.



**Project>Options>C/C++ Compiler>Align functions**

## --char\_is\_signed

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--char_is_signed</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description | <p>By default, the compiler interprets the <code>char</code> type as unsigned. Use this option to make the compiler interpret the <code>char</code> type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.</p> <p><b>Note:</b> The runtime library is compiled without the <code>--char_is_signed</code> option. If you use this option, you might get type mismatch warnings from the linker, because the library uses unsigned <code>char</code>.</p> |



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

## --code\_model

|                            |                                                                                                                                                                                                                                                 |                            |                                                                          |                   |                                                    |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|--------------------------------------------------------------------------|-------------------|----------------------------------------------------|
| Syntax                     | <code>--code_model={far f huge h}</code>                                                                                                                                                                                                        |                            |                                                                          |                   |                                                    |
| Parameters                 | <table> <tr> <td><code>far</code> (default)</td><td>Functions and constant data can be placed in the high 8 Mbytes of memory</td></tr> <tr> <td><code>huge</code></td><td>Functions and constant data can be placed anywhere</td></tr> </table> | <code>far</code> (default) | Functions and constant data can be placed in the high 8 Mbytes of memory | <code>huge</code> | Functions and constant data can be placed anywhere |
| <code>far</code> (default) | Functions and constant data can be placed in the high 8 Mbytes of memory                                                                                                                                                                        |                            |                                                                          |                   |                                                    |
| <code>huge</code>          | Functions and constant data can be placed anywhere                                                                                                                                                                                              |                            |                                                                          |                   |                                                    |
| Description                | Use this option to select the code model for which the code will be generated. If you do not select a code model option, the compiler uses the default code model.                                                                              |                            |                                                                          |                   |                                                    |
| See also                   | <i>Code models and memory attributes for function storage</i> , page 21.                                                                                                                                                                        |                            |                                                                          |                   |                                                    |



**Project>Options>General Options>Target>Code model**

## -D

|                     |                                                                                                                                                                                        |                     |                                     |                    |                                      |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|-------------------------------------|--------------------|--------------------------------------|
| Syntax              | <code>-D symbol[=value]</code>                                                                                                                                                         |                     |                                     |                    |                                      |
| Parameters          | <table> <tr> <td><code>symbol</code></td><td>The name of the preprocessor symbol</td></tr> <tr> <td><code>value</code></td><td>The value of the preprocessor symbol</td></tr> </table> | <code>symbol</code> | The name of the preprocessor symbol | <code>value</code> | The value of the preprocessor symbol |
| <code>symbol</code> | The name of the preprocessor symbol                                                                                                                                                    |                     |                                     |                    |                                      |
| <code>value</code>  | The value of the preprocessor symbol                                                                                                                                                   |                     |                                     |                    |                                      |
| Description         | Use this option to define a preprocessor symbol. If no value is specified, 1 is used. This option can be used one or more times on the command line.                                   |                     |                                     |                    |                                      |

The option `-D` has the same effect as a `#define` statement at the top of the source file:

`-Dsymbol`

is equivalent to:

`#define symbol 1`

To get the equivalence of:

`#define FOO`

specify the `=` sign but nothing after, for example:

`-DFOO=`



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

## --data\_model

Syntax

`--data_model={near|n|far|f|huge|h}`

Parameters

|                            |                                                               |
|----------------------------|---------------------------------------------------------------|
| <code>near</code>          | Places variables in the lowest or highest 32 Kbytes of memory |
| <code>far</code> (default) | Places variables in the lowest or highest 8 Mbytes of memory  |
| <code>huge</code>          | Places variables anywhere in memory                           |

Description

Use this option to select the data model for which the code will be generated. If you do not select a data model option, the compiler uses the default data model.

See also

*Data models*, page 12.



**Project>Options>General Options>Target>Data model**

## --debug, -r

Syntax

`--debug`  
`-r`

Description

Use the `--debug` or `-r` option to make the compiler include information in the object modules required by the IAR C-SPY® Debugger and other symbolic debuggers.

**Note:** Including debug information will make the object files larger than otherwise.



**Project>Options>C/C++ Compiler>Output>Generate debug information**



# --dependencies

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

## Parameters

|             |                               |
|-------------|-------------------------------|
| i (default) | Lists only the names of files |
| m           | Lists in makefile style       |

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 126.

## Description

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

## Example

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

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

If `--dependencies=m` is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:

```
foo.r53: c:\iar\product\include\stdio.h
foo.r53: 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:

```
%.r53 : %.c
$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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



This option is not available in the IDE.

**--diag\_error**

|             |                                                                                                                                                                                                                                                                                          |                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| Syntax      | <code>--diag_error=tag[, tag, ...]</code>                                                                                                                                                                                                                                                |                                                                          |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                               | The number of a diagnostic message, for example the message number Pe117 |
| Description | Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated. The exit code will be non-zero. This option may be used more than once on the command line. |                                                                          |



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

**--diag\_remark**

|             |                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                          |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| Syntax      | <code>--diag_remark=tag[, tag, ...]</code>                                                                                                                                                                                                                                                                                                                                                                  |                                                                          |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                                                                                                                                                  | The number of a diagnostic message, for example the message number Pe177 |
| Description | Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line.<br><br><b>Note:</b> By default, remarks are not displayed; use the <code>--remarks</code> option to display them. |                                                                          |



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks**

**--diag\_suppress**

|            |                                              |                                                                          |
|------------|----------------------------------------------|--------------------------------------------------------------------------|
| Syntax     | <code>--diag_suppress=tag[, tag, ...]</code> |                                                                          |
| Parameters | <i>tag</i>                                   | The number of a diagnostic message, for example the message number Pe117 |

**Description** Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.



**Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics**

**--diag\_warning**

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

**Parameters**

|            |                                                                          |
|------------|--------------------------------------------------------------------------|
| <i>tag</i> | The number of a diagnostic message, for example the message number Pe826 |
|------------|--------------------------------------------------------------------------|

**Description** Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings**

**--diagnostics\_tables**

**Syntax** `--diagnostics_tables {filename|directory}`

**Parameters** For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 208.

**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 | <p>Use this option to discard unused public functions and variables from the compilation unit. This enhances interprocedural optimizations such as inlining, cross call, and cross jump by limiting their scope to public functions and variables that are actually used.</p> <p>This option is only useful when <i>all</i> source files are compiled as one unit, which means that the <code>--mfc</code> compiler option is used.</p> <p><b>Note:</b> Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output.</p> |
| See also    | <code>--mfc</code> , page 141 and <i>Multi-file compilation units</i> , page 107.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |



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

# **--dlib\_config**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--dlib_config filename</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 126.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Description | <p>Each runtime library has a corresponding library configuration file. Use this option to specify the library configuration file for the compiler. Make sure that you specify a configuration file that corresponds to the library you are using.</p> <p>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 <code>r32c\lib</code>. For examples and a list of prebuilt runtime libraries, see <i>Using a prebuilt library</i>, page 44.</p> <p>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 <i>Building and using a customized library</i>, page 50.</p> |



To set related options, choose:

**Project>Options>General Options>Library Configuration**

--double

|             |                                                                                                                                                                                                                                                                   |                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| Syntax      | --double={32   64}                                                                                                                                                                                                                                                |                         |
| Parameters  | 32 (default)                                                                                                                                                                                                                                                      | 32-bit doubles are used |
|             | 64                                                                                                                                                                                                                                                                | 64-bit doubles are used |
| Description | Use this option to select the precision used by the compiler for representing the floating-point types <code>double</code> and <code>long double</code> . The compiler can use either 32-bit or 64-bit precision. By default, the compiler uses 32-bit precision. |                         |
| See also    | Floating-point types, page 156.                                                                                                                                                                                                                                   |                         |



**Project>Options>General Options>Target>Size of type 'double'**

-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.<br><br><b>Note:</b> The <code>-e</code> option and the <code>--strict_ansi</code> option cannot be used at the same time. |
| See also    | The chapter <i>Compiler extensions</i> .                                                                                                                                                                                                                                                                                                                                    |



**Project>Options>C/C++ Compiler>Language>Allow IAR extensions**

**Note:** By default, this option is enabled 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>Embedded C++**

**--eec++**

|             |                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --eec++                                                                                                                                                                                                                                                            |
| Description | In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++. |
| See also    | <i>Extended Embedded C++</i> , page 92.                                                                                                                                                                                                                            |



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

**--enable\_multibytes**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --enable_multibytes                                                                                                                                                                                                                                                                                                                                                                                               |
| Description | <p>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.</p> <p>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.</p> |



**Project>Options>C/C++ Compiler>Language>Enable multibyte support**


**--error\_limit**

|             |                                                                                                                                                                              |          |                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --error_limit= <i>n</i>                                                                                                                                                      |          |                                                                                                                            |
| 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 --error_limit option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.                          |          |                                                                                                                            |




This option is not available in the IDE.

**-f**

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>-f filename</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Parameters   | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 208.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Descriptions | <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> <p> To set this option, use <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b>.</p> |

**--fp\_model**

|                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                             |                                                                                        |                   |                                                                               |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|----------------------------------------------------------------------------------------|-------------------|-------------------------------------------------------------------------------|
| Syntax                      | <code>--fp_model={fast   full}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                             |                                                                                        |                   |                                                                               |
| Parameters                  | <table><tr><td><code>fast</code> (default)</td><td>The compiler will insert fast, but non-standard, FPU instructions directly in the code</td></tr><tr><td><code>full</code></td><td>The compiler will use standard IEEE 754-compliant floating-point instructions</td></tr></table>                                                                                                                                                                                                                                                                                                                                   | <code>fast</code> (default) | The compiler will insert fast, but non-standard, FPU instructions directly in the code | <code>full</code> | The compiler will use standard IEEE 754-compliant floating-point instructions |
| <code>fast</code> (default) | The compiler will insert fast, but non-standard, FPU instructions directly in the code                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                             |                                                                                        |                   |                                                                               |
| <code>full</code>           | The compiler will use standard IEEE 754-compliant floating-point instructions                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                             |                                                                                        |                   |                                                                               |
| Description                 | <p>Use this option to control the type of floating-point instructions that the compiler uses. The more efficient Fast model requires a hardware floating-point unit (FPU). It is not compliant with the IEEE-754 standard and does not support, for example, Infinity and NaN.</p> <p>The Full model is less efficient but fully supports the IEEE-754 standard. If the device is equipped with an FPU, the Full model can use standards-compliant library functions but still use the FPU instructions. If the device does not have an FPU, library routines that emulate floating-point operations will be used.</p> |                             |                                                                                        |                   |                                                                               |
| See also                    | <p><i>Floating-point implementation</i>, page 64.</p> <p> To set related options, choose:<br/><b>Project&gt;Options&gt;General Options&gt;Target&gt;Float implementation</b></p>                                                                                                                                                                                                                                                                                                                                                    |                             |                                                                                        |                   |                                                                               |

**--header\_context**

|             |                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --header_context                                                                                                                                                                                                                                                                     |
| Description | Occasionally, to find the cause of a problem it is necessary to know which header file that was included from which source line. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point. |



This option is not available in the IDE.

**-I**

|             |                                                                                                                             |
|-------------|-----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | -I path                                                                                                                     |
| Parameters  | path                      The search path for #include files                                                                |
| Description | Use this option to specify the search paths for #include files. This option can be used more than once on the command line. |
| See also    | Include file search procedure, page 120.                                                                                    |



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

**-l**

|            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |                     |   |                                                      |   |                                                                                                                                                                                                                                           |   |                                                                                                                                                                                                                                           |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---------------------|---|------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | -l[a A b B c C D][N][H] {filename directory}                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |   |                     |   |                                                      |   |                                                                                                                                                                                                                                           |   |                                                                                                                                                                                                                                           |
| Parameters | <table><tr><td>a</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 -la, 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 -lA, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *</td></tr></table> | a | Assembler list file | A | Assembler list file with C or C++ source as comments | b | Basic assembler list file. This file has the same contents as a list file produced with -la, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included * | B | Basic assembler list file. This file has the same contents as a list file produced with -lA, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included * |
| a          | Assembler list file                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |   |                     |   |                                                      |   |                                                                                                                                                                                                                                           |   |                                                                                                                                                                                                                                           |
| A          | Assembler list file with C or C++ source as comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |   |                     |   |                                                      |   |                                                                                                                                                                                                                                           |   |                                                                                                                                                                                                                                           |
| b          | Basic assembler list file. This file has the same contents as a list file produced with -la, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *                                                                                                                                                                                                                                                                                                                                                                                                                                                |   |                     |   |                                                      |   |                                                                                                                                                                                                                                           |   |                                                                                                                                                                                                                                           |
| B          | Basic assembler list file. This file has the same contents as a list file produced with -lA, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *                                                                                                                                                                                                                                                                                                                                                                                                                                                |   |                     |   |                                                      |   |                                                                                                                                                                                                                                           |   |                                                                                                                                                                                                                                           |



|             |                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|
| a           | Assembler list file                                                                                                                |
| 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 126.

**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**

**--mfc**

**Syntax** --mfc

**Description** Use this option to enable *multi-file compilation*. This means that the compiler compiles one or several source files specified on the command line as one unit, which makes interprocedural optimizations such as inlining, cross call, and cross jump possible.

**Note:** The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the `-o` compiler option and specify a certain output file.

|          |                                                                                                                                                           |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Example  | <code>iccr32c myfile1.c myfile2.c myfile3.c --mfc</code>                                                                                                  |
| See also | <code>--discard_unused_publics</code> , page 136, <code>-o</code> , <code>--output</code> , page 147, and <i>Multi-file compilation units</i> , page 107. |



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

**--migration\_preprocessor\_extensions**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--migration_preprocessor_extensions</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Description | <p>If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you might want to use this option. Use this option to use the following in preprocessor expressions:</p> <ul style="list-style-type: none"><li>● Floating-point expressions</li><li>● Basic type names and <code>sizeof</code></li><li>● All symbol names (including typedefs and variables).</li></ul> <p><b>Note:</b> If you use this option, not only will the compiler accept code that does not conform to the ISO/ANSI C standard, but it will also reject some code that <i>does</i> conform to the standard.</p> <p><b>Important!</b> Do not depend on these extensions in newly written code, because support for them might be removed in future compiler versions.</p> |



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

**--module\_name**

|            |                                 |                                |
|------------|---------------------------------|--------------------------------|
| Syntax     | <code>--module_name=name</code> |                                |
| Parameters | <i>name</i>                     | An explicit object module name |

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>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.</p> <p>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.</p> |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



**Project>Options>C/C++ Compiler>Output>Object module name**

## --no\_code\_motion

|             |                                                                                                                                                                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_code_motion</code>                                                                                                                                                                                                                                                                                                               |
| Description | <p>Use this option to disable code motion optimizations. These optimizations, which are performed at the optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code might be difficult to debug.</p> <p><b>Note:</b> This option has no effect at optimization levels below Medium.</p> |



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

## --no\_cross\_call

|             |                                                                                                                                                                                                                                                |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_cross_call</code>                                                                                                                                                                                                                   |
| Description | <p>Use this option to disable the cross-call optimization. This optimization is performed at size optimization, level High. Note that, although the option can drastically reduce the code size, this option increases the execution time.</p> |



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Cross call**

## --no\_cse

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

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



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination**

## **--no\_inline**

Syntax

`--no_inline`

Description

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

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

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

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



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

## **--no\_path\_in\_file\_macros**

Syntax

`--no_path_in_file_macros`

Description

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

See also

*Descriptions of predefined preprocessor symbols*, page 212.



This option is not available in the IDE.

## --no\_tbaa

|             |                                                                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_tbaa</code>                                                                                                                                                                                 |
| Description | Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through unsigned char. |
| See also    | <i>Type-based alias analysis</i> , page 110.                                                                                                                                                           |



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

## --no\_typedefs\_in\_diagnostics

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_typedefs_in_diagnostics</code>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Description | Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.                                                                                                                                             |
| Example     | <pre>typedef int (*MyPtr)(char const *); MyPtr p = "foo";</pre> <p>will give an error message like this:</p> <pre>Error[Pe144]: a value of type "char *" cannot be used to initialize an entity of type "MyPtr"</pre> <p>If the <code>--no_typedefs_in_diagnostics</code> option is used, the error message will be like this:</p> <pre>Error[Pe144]: a value of type "char *" cannot be used to initialize an entity of type "int (*)(char const *)"</pre> |



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

## --no\_unroll

Syntax

--no\_unroll

Description

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

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

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

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

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



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling**

## --no\_warnings

Syntax

--no\_warnings

Description

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



This option is not available in the IDE.

## --no\_wrap\_diagnostics

Syntax

--no\_wrap\_diagnostics

Description

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



This option is not available in the IDE.

-O

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |                            |             |      |   |        |   |                |    |                      |    |                     |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|----------------------------|-------------|------|---|--------|---|----------------|----|----------------------|----|---------------------|
| Syntax      | -O [n   l   m   h   hs   hz]                                                                                                                                                                                                                                                                                                                                                                                                                                             |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| Parameters  | <table><tr><td>n</td><td>None* (Best debug support)</td></tr><tr><td>l (default)</td><td>Low*</td></tr><tr><td>m</td><td>Medium</td></tr><tr><td>h</td><td>High, balanced</td></tr><tr><td>hs</td><td>High, favoring speed</td></tr><tr><td>hz</td><td>High, favoring size</td></tr></table>                                                                                                                                                                             | n | None* (Best debug support) | l (default) | Low* | m | Medium | h | High, balanced | hs | High, favoring speed | hz | High, favoring size |
| n           | None* (Best debug support)                                                                                                                                                                                                                                                                                                                                                                                                                                               |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| l (default) | Low*                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| m           | Medium                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| h           | High, balanced                                                                                                                                                                                                                                                                                                                                                                                                                                                           |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| hs          | High, favoring speed                                                                                                                                                                                                                                                                                                                                                                                                                                                     |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| hz          | High, favoring size                                                                                                                                                                                                                                                                                                                                                                                                                                                      |   |                            |             |      |   |        |   |                |    |                      |    |                     |
|             | <b>*The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.</b>                                                                                                                                                                                                                                                                                                                                |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| Description | <p>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.</p> <p>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.</p> |   |                            |             |      |   |        |   |                |    |                      |    |                     |
| See also    | <i>Controlling compiler optimizations</i> , page 106.                                                                                                                                                                                                                                                                                                                                                                                                                    |   |                            |             |      |   |        |   |                |    |                      |    |                     |



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


-o, --output

|             |                                                                                                                                                                                                                                                                  |  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>-o {filename directory}</code><br><code>--output {filename directory}</code>                                                                                                                                                                               |  |
| Parameters  | For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 208.                                                                                                                    |  |
| 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>.obj</code> . Use this option to explicitly specify a different output filename for the object code output. |  |




This option is not available in the IDE.


# --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 <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b> .                                                                                                                                                                                                                                                                                             |

# --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.                                     |

# -o, --output


|             |                                                                                                                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | -o {filename directory}<br>--output {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 208.                                                                                                     |
| 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 r53. 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 126.                                                                                                                                                                                                                                                   |
| Description | <p>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.</p> <p>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.</p> |
|             |  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 126.                                                                                                                                   |
| Description | Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol. |
|             |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Preprocessor&gt;Preinclude file</b>                                                                                               |

# --preprocess

|            |                                                                                                                                                                     |   |                   |   |                 |   |                           |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|-------------------|---|-----------------|---|---------------------------|
| Syntax     | <code>--preprocess [= [c] [n] [l]] {filename directory}</code>                                                                                                      |   |                   |   |                 |   |                           |
| Parameters | <table> <tr> <td>c</td><td>Preserve comments</td></tr> <tr> <td>n</td><td>Preprocess only</td></tr> <tr> <td>l</td><td>Generate #line directives</td></tr> </table> | c | Preserve comments | n | Preprocess only | l | Generate #line directives |
| c          | Preserve comments                                                                                                                                                   |   |                   |   |                 |   |                           |
| n          | Preprocess only                                                                                                                                                     |   |                   |   |                 |   |                           |
| l          | Generate #line directives                                                                                                                                           |   |                   |   |                 |   |                           |
|            | For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 126.                       |   |                   |   |                 |   |                           |

|                     |                                                                                                                                                                                                                                |                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Description         | Use this option to generate preprocessed output to a named file.                                                                                                                                                               |                                                                                            |
|                     |                                                                                                                                               | <b>Project&gt;Options&gt;C/C++ Compiler&gt;Preprocessor&gt;Preprocessor output to file</b> |
| <b>--public_equ</b> |                                                                                                                                                                                                                                |                                                                                            |
| Syntax              | <code>--public_equ <i>symbol</i>[=<i>value</i>]</code>                                                                                                                                                                         |                                                                                            |
| Parameters          | <i>symbol</i>                                                                                                                                                                                                                  | The name of the assembler symbol to be defined                                             |
|                     | <i>value</i>                                                                                                                                                                                                                   | An optional value of the defined assembler symbol                                          |
| Description         | This option is equivalent to defining a label in assembler language using the <code>EQU</code> directive and exporting it using the <code>PUBLIC</code> directive. This option can be used more than once on the command line. |                                                                                            |
|                     |                                                                                                                                               | This option is not available in the IDE.                                                   |

**-r, --debug**

|             |                                                                                                                                                                                                                                                                                              |                                                                                     |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| Syntax      | <code>-r</code><br><code>--debug</code>                                                                                                                                                                                                                                                      |                                                                                     |
| Description | Use the <code>-r</code> or the <code>--debug</code> option to make the compiler include information in the object modules required by the IAR C-SPY Debugger and other symbolic debuggers.<br><br><b>Note:</b> Including debug information will make the object files larger than otherwise. |                                                                                     |
|             |                                                                                                                                                                                                           | <b>Project&gt;Options&gt;C/C++ Compiler&gt;Output&gt;Generate debug information</b> |

**--remarks**

|             |                                                                                                                                                                                                                                                                       |  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--remarks</code>                                                                                                                                                                                                                                                |  |
| Description | The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks. |  |

See also *Severity levels*, page 204.



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

## --require\_prototypes

Syntax `--require_prototypes`

Description Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

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



**Project>Options>C/C++ Compiler>Language>Require prototypes**

## --silent

Syntax `--silent`

Description By default, the compiler issues introductory messages and a final statistics report. Use this option to make the compiler operate without sending these messages to the standard output stream (normally the screen).

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



This option is not available in the IDE.

## --strict\_ansi

Syntax `--strict_ansi`

Description By default, the compiler accepts a relaxed superset of ISO/ANSI C/C++, see *Minor language extensions*, page 169. Use this option to ensure that the program conforms to the ISO/ANSI C/C++ standard.

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



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

### **--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. |

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

See also *diag\_warning*, page 296.



**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` or the `__packed` data type attribute.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

### ALIGNMENT ON THE R32C/100 MICROCOMPUTER

The R32C/100 microcomputer can access memory using 8- to 64-bit operations. However, when an unaligned access is performed, more bus cycles are required. The compiler avoids this by assigning an alignment to every data type.

# Basic data types

The compiler supports both all ISO/ANSI C basic data types and some additional types.

## INTEGER TYPES

This table gives the size and range of each integer data type:

| Data type          | Size    | Range                                  | Alignment |
|--------------------|---------|----------------------------------------|-----------|
| bool               | 8 bits  | 0 to 1                                 | 1         |
| char               | 8 bits  | 0 to 255                               | 1         |
| signed char        | 8 bits  | -128 to 127                            | 1         |
| unsigned char      | 8 bits  | 0 to 255                               | 1         |
| signed short       | 16 bits | -32768 to 32767                        | 2         |
| unsigned short     | 16 bits | 0 to 65535                             | 2         |
| signed int         | 32 bits | -2 <sup>31</sup> to 2 <sup>31</sup> -1 | 4         |
| unsigned int       | 32 bits | 0 to 2 <sup>32</sup> -1                | 4         |
| signed long        | 32 bits | -2 <sup>31</sup> to 2 <sup>31</sup> -1 | 4         |
| unsigned long      | 32 bits | 0 to 2 <sup>32</sup> -1                | 4         |
| signed long long   | 64 bits | -2 <sup>63</sup> to 2 <sup>63</sup> -1 | 8         |
| unsigned long long | 64 bits | 0 to 2 <sup>64</sup> -1                | 8         |

Table 32: Integer types

Signed variables are represented using the two’s complement form.

## Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

## The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring signed rather than unsigned.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example,

```
/* 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.

## Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the IAR C/C++ Compiler for R32C, any integer type can be used as the base type when language extensions are enabled.

Bitfields in expressions will have the same data type as the integer base type.

By default, the compiler places bitfield members from the least significant to the most significant bit in the container type.

If you use the directive `#pragma bitfields=reversed`, the bitfield members are placed from the most significant to the least significant bit.

FLOATING-POINT TYPES

In the IAR C/C++ Compiler for R32C, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

| Type        | Size if --double=32 | Size if --double=64 |
|-------------|---------------------|---------------------|
| float       | 32 bits             | 32 bits             |
| double      | 32 bits (default)   | 64 bits             |
| long double | 32 bits             | 64 bits             |

Table 33: Floating-point types

**Note:** The size of `double` and `long double` depends on the `--double={32|64}` option, see `--double`, page 137. The type `long double` uses the same precision as `double`.

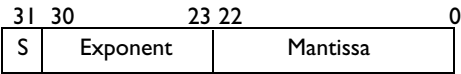
The compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero.

Exception flags according to the IEEE 754 standard are not supported.

See also *Floating-point implementation*, page 64.

32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:



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

The value of the number is:

$(-1)^S * 2^{(Exponent-127)} * 1.Mantissa$

The range of the number is:

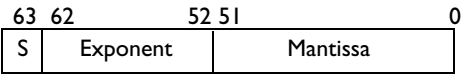
$\pm 1.18E-38 \text{ to } \pm 3.39E+38$

The precision of the float operators (+, -, \*, and /) is approximately 7 decimal digits.



### 64-bit floating-point format

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



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

$$(-1)^S * 2^{(Exponent-1023)} * 1.Mantissa$$

The range of the number is:

$$\pm 2.23E-308 \text{ to } \pm 1.79E+308$$

The precision of the float operators (+, -, \*, and /) is approximately 15 decimal digits.

### Representation of special floating-point numbers

This list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.
- If you are using the Fast floating-point model (`--fp_model=fast`), Not a number (NaN) and Infinity are not recognized. See *Floating-point implementation*, page 64.

## Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

### FUNCTION POINTERS

The function pointer of the R32C IAR C/C++ Compiler is `__code32`. It is a 32-bit pointer that can address the entire memory. The internal representation of the function pointer is the actual address it refers to.

## DATA POINTERS

The data pointer of the R32C IAR C/C++ Compiler is `__data32`. It is a 32-bit signed int pointer that can address the entire memory.

## 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 *value* of an signed integer type to a pointer of a larger type is performed by signed extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result.

### size\_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the IAR C/C++ Compiler for R32C, the size of `size_t` is 32 bits.

### ptrdiff\_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the IAR C/C++ Compiler for R32C, the size of `ptrdiff_t` is 32 bits.

### intptr\_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for R32C, the size of `intptr_t` is 32 bits.

### uintptr\_t

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned.

---

## Structure types

The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

## ALIGNMENT

The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. The size of a `struct` is also adjusted to allow arrays of aligned structure objects.

## GENERAL LAYOUT

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

### Example

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

This diagram shows the layout in memory:

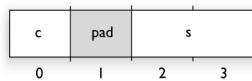


Figure 6: Structure layout

The alignment of the structure is 2 bytes, and a pad byte must be inserted to give `short s` the correct alignment.

## PACKED STRUCTURE TYPES

The `#pragma pack` attribute or the `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is 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.

**Example**

This example declares a packed structure:

```
#pragma pack(1)
struct S {
 char c;
 short s;
};
```

```
#pragma pack()
```

In this example, the structure `s` has this memory layout:



Figure 7: Packed structure layout

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

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

`S2` has this memory layout

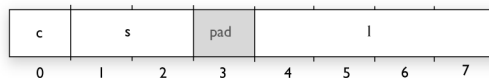


Figure 8: Packed structure layout

The structure `s` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 4, which means that alignment of the structure `S2` will become 4.

For more information, see *Alignment of elements in a structure*, page 101.

## Type qualifiers

According to the ISO/ANSI C standard, `volatile` and `const` are type qualifiers.

### DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

### Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for R32C are described below.

### Rules for accesses

In the IAR C/C++ Compiler for R32C, 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 all 8-bit data types and for 16- and 32-bit types if they are aligned.

For all other object types, only the rule that states that all accesses are preserved applies.

## DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` and located in the memories `data16`, `data24`, and `data32` are allocated in ROM. For `sbdata16` and `sbdata24`, the objects are allocated in RAM and initialized by the runtime system at startup.

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.

# Compiler extensions

This chapter gives a brief overview of the compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

---

## Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

For a summary of available language extensions, see *C language extensions*, page 164. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages.

Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For a list of available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

The preprocessor of the compiler adheres to the ISO/ANSI standard. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of

instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 71. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. The library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 218.

**Note:** Any use of these extensions, except for the pragma directives, makes your application inconsistent with the ISO/ANSI C standard.

## ENABLING LANGUAGE EXTENSIONS



In the IDE, language extensions are enabled by default.

For information about how to enable and disable language extensions from the command line, see the compiler options `-e`, page 137, and `--strict_ansi`, page 151.

---

## C language extensions

This section gives a brief overview of the C language extensions available in the compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions are grouped according to their expected usefulness. In short, this means:

- Important language extensions—extensions specifically tailored for efficient embedded programming, typically to meet memory restrictions
- Useful language extensions—features considered useful and typically taken from related standards, such as C99 and C++
- Minor language extensions, that is, the relaxation of some minor standards issues and also some useful but minor syntax extensions.

## IMPORTANT LANGUAGE EXTENSIONS

The following language extensions available both in the C and the C++ programming languages are well suited for embedded systems programming:

- 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 primitives, see *Controlling data and function placement in memory*, page 103, and *location*, page 192.

- Alignment

Each data type has its own alignment, for more details, see *Alignment*, page 153. If you want to change the alignment, the `__packed` data type attribute, and the `#pragma pack` and `#pragma data_alignment` directives are available. If you want to use the alignment of an object, use the `__ALIGNOF__()` operator.

The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

- `__ALIGNOF__(type)`
- `__ALIGNOF__(expression)`

In the second form, the expression is not evaluated.

- Anonymous structs and unions

C++ includes a feature named anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 101.

- Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of type `int` or `unsigned int`. Using IAR Systems language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 155.

- Dedicated segment operators `__segment_begin` and `__segment_end`

The syntax for these operators is:

```
void * __segment_begin(segment)
void * __segment_end(segment)
```

These operators return the address of the first byte of the named *segment* and the first byte *after* the named *segment*, respectively. This can be useful if you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment.

The named *segment* must be a string literal and *segment* 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.

In this example, the type of the `__segment_begin` operator is `void __huge *`.

```
#pragma segment="MYSEGMENT" __huge
...
segment_start_address = __segment_begin("MYSECTION");
```

See also *segment*, page 197, and *location*, page 192.

## USEFUL LANGUAGE EXTENSIONS

This section lists and briefly describes useful extensions, that is, useful features typically taken from related standards, such as C99 and C++:

- **Inline functions**

The `#pragma inline` directive, alternatively the `inline` keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword `inline`. For more information, see *inline*, page 191.

- **Mixing declarations and statements**

It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.

- **Declaration in `for` loops**

It is possible to have a declaration in the initialization expression of a `for` loop, for example:

```
for (int i = 0; i < 10; ++i)
{ ... }
```

This feature is part of the C99 standard and C++.

- **The `bool` data type**

To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)

- **C++ style comments**

C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

```
// The length of the bar, in centimeters.
int length;
```

This feature is copied from the C99 standard and C++.

### Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict-ansi` is used. The `__asm` keyword is always available.

**Note:** Not all assembler directives or operators can be inserted using this keyword.

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"
 " jmp Label");
```

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

For more information about inline assembler, see *Mixing C and assembler*, page 71.

## Compound literals

To create compound literals you can use this syntax:

```
/* Create a pointer to an anonymous array */
int *p = (int []) {1,2,3};

/* Create a pointer to an anonymous structX */
structX *px = &(amp;structX) {5,6,7};
```

### Note:

- A compound literal can be modified unless it is declared `const`.
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

## Incomplete arrays at end of structs

The last element of a `struct` can be an incomplete array. This is useful for allocating a chunk of memory that contains both the structure and a fixed number of elements of the array. The number of elements can vary between allocations.

This feature is part of the C99 standard.

**Note:** The array cannot be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

### Example

```
struct str
{
 char a;
 unsigned long b[];
};
```

```

struct str * GetAStr(int size)
{
 return malloc(sizeof(struct str) +
 sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
 s->b[10] = 0;
}

```

The incomplete array will be aligned in the structure just like any other member of the structure. For more information about structure alignment, see *Structure types*, page 158.

### Hexadecimal floating-point constants

Floating-point constants can be given in hexadecimal style. The syntax is  $0xMANTp\{+|- \}EXP$ , where *MANT* is the mantissa in hexadecimal digits, including an optional . (decimal point), and *EXP* is the exponent with decimal digits, representing an exponent of 2. This feature is part of the C99 standard.

#### Examples

$0x1p0$  is 1

$0xA.8p2$  is  $10.5 \times 2^2$

### Designated initializers in structures and arrays

Any initialization of either a structure (struct or union) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as *.elementname* and for an array [*constant index expression*]. Using designated initializers is not supported in C++.

### Examples

This definition shows a `struct` and its initialization using designators:

```
struct{
 int i;
 int j;
 int k;
 int l;
 short array[10];
} u = {
 .l = 6, /* initialize l to 6 */
 .j = 6, /* initialize j to 6 */
 8, /* initialize k to 8 */
 .array[7] = 2, /* initialize element 7 to 2 */
 .array[3] = 2, /* initialize element 3 to 2 */
 5, /* array[4] = 5 */
 .k = 4 /* reinitialize k to 4 */
};
```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```
union {
 int i;
 float f;
} y = {.f = 5.0};
```

To set the size of an array by initializing the last element, do like this:

```
char array[] = {[10] = 'a'};
```

## MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards issues and also some useful but minor syntax extensions:

- **Arrays of incomplete types**  
An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).
- **Forward declaration of `enum` types**  
The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- Missing semicolon at end of `struct` or `union` specifier

A warning is issued if the semicolon at the end of a `struct` or `union` specifier is missing.

- Null and `void`

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

- Casting pointers to integers in static initializers

In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 158.

- Taking the address of a register variable

In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.

- Duplicated size and sign specifiers

Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

- `long float` means `double`

The type `long float` is accepted as a synonym for `double`.

- Repeated `typedef` declarations

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.

- Non-top level `const`

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`).

Comparing and taking the difference of such pointers is also allowed.

- Non-lvalue arrays

A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.

- Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.

- An extra comma at the end of `enum` lists

Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

- A label preceding a `}`

In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the IAR C/C++ Compiler for R32C, a warning is issued.

**Note:** This also applies to the labels of `switch` statements.

- Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued. In the IAR C/C++ Compiler for R32C, this expression is allowed:

```
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, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char) "
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 137.



# Extended keywords

This chapter describes the extended keywords that support specific features of the R32C/100 microcomputer 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 R32C/100 microcomputer. 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 information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 177.

**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 137 for additional information.

### TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

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 microcomputer.

- Available *function memory attributes*: `__code24` and `__code32`
- Available *data memory attributes*: `__data16`, `__data24`, `__data32`, `__sdata16`, and `__sdata24`

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 used. 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`, `__task`, and `__fast_interrupt`
- *Data type attributes*: `const`, `__packed`, and `volatile`

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

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 161.

## 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 `__data24` memory attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in data24 memory. The variables `k` and `l` behave in the same way:

```
__data24 int i, j;
int __data24 k, l;
```

Note that the attribute affects both identifiers.

This declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__data24
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 16.

An easier way of specifying storage is to use type definitions. These two declarations are equivalent:

```
typedef char __data24 Byte;
Byte b;
```

and

```
__data24 char b;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

### 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);
```

### Syntax for type attributes on function pointers

To declare a function pointer, use this syntax:

```
int (__code32 * fp) (double);
```

After this declaration, the function pointer `fp` points to code32 memory.

An easier way of specifying storage is to use type definitions:

```
typedef __code32 void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
```

```
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

## OBJECT ATTRIBUTES

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__monitor`, `__noreturn`, 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 103. For more information about `vector`, see *vector*, page 198.

### 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>__code24</code>         | Controls the storage of functions                                                 |
| <code>__code32</code>         | Controls the storage of functions                                                 |
| <code>__data16</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>__fast_interrupt</code> | Supports fast interrupt functions                                                 |
| <code>__interrupt</code>      | Supports interrupt functions                                                      |
| <code>__intrinsic</code>      | Reserved for compiler internal use only                                           |
| <code>__monitor</code>        | Supports atomic execution of a function                                           |
| <code>__no_init</code>        | Supports non-volatile memory                                                      |
| <code>__noreturn</code>       | Informs the compiler that the declared function will not return                   |
| <code>__packed</code>         | Decreases data type alignment to 1                                                |
| <code>__root</code>           | Ensures that a function or variable is included in the object code even if unused |
| <code>__sdata16</code>        | Controls the storage of data objects                                              |
| <code>__sdata24</code>        | Controls the storage of data objects                                              |
| <code>__task</code>           | Allows functions to exit without restoring registers                              |

Table 34: Extended keywords summary

# Descriptions of extended keywords

These sections give detailed information about each extended keyword.

## `__code24`

|                     |                                                                                                                                                                          |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 173.                                        |
| Description         | The <code>__code24</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in code24 memory. |
| Storage information | <ul style="list-style-type: none"><li>Address range: 0xFF800000–0xFFFFFFFF (8 Mbytes)</li></ul>                                                                          |

- Maximum size: 8 Mbytes
- Pointer size: 4 bytes

Example

```
__code24 void myfunction(void);
```

See also

*Code models and memory attributes for function storage*, page 21.

## **\_\_code32**

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 173.

Description

The `__code32` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in code32 memory.

Storage information

- Address range: 0–0xFFFFFFFF (4 Gbytes)
- Maximum size: 4 Gbytes
- Pointer size: 4 bytes

Example

```
__code32 void myfunction(void);
```

See also

*Code models and memory attributes for function storage*, page 21.

## **\_\_data16**

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 173.

Description

The `__data16` memory attribute overrides the default storage of variables and constants given by the selected data model and code model, respectively, and places individual variables and constants in data16 memory.

Storage information

- Address range: 0–0x7FFF, 0xFFFF8000–0xFFFFFFFF (32 Kbytes)
- Maximum object size: 32 Kbytes
- Pointer size: 4 bytes.

Example

```
__data16 int x;
```

See also

*Memory types*, page 13.

## **\_\_data24**

|                     |                                                                                                                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 173.                                                                                               |
| Description         | The <code>__data24</code> memory attribute overrides the default storage of variables and constants given by the selected data model and code model, respectively, and places individual variables and constants in data24 memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0–0x7FFFFF, 0xFF800000–0xFFFFFFFF (8 Mbytes)</li> <li>● Maximum object size: 8 Mbytes–1</li> <li>● Pointer size: 4 bytes</li> </ul>                                        |
| Example             | <code>__data24 int x;</code>                                                                                                                                                                                                       |
| See also            | <i>Memory types</i> , page 13.                                                                                                                                                                                                     |

## **\_\_data32**

|                     |                                                                                                                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 173.                                                                                               |
| Description         | The <code>__data32</code> memory attribute overrides the default storage of variables and constants given by the selected data model and code model, respectively, and places individual variables and constants in data32 memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0–0xFFFFFFFF (4 Gbytes)</li> <li>● Maximum object size: 2 Gbytes–1</li> <li>● Pointer size: 4 bytes.</li> </ul>                                                            |
| Example             | <code>__data32 int x;</code>                                                                                                                                                                                                       |
| See also            | <i>Memory types</i> , page 13.                                                                                                                                                                                                     |

## **\_\_fast\_interrupt**

|             |                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 173.                                                             |
| Description | The <code>__fast_interrupt</code> keyword specifies a very fast interrupt function of the highest priority, using the FREIT return mechanism. The interrupt uses the VCT register as a |

vector. The register is set up during early initialization and will point to the fast interrupt function in the application, if one has been declared.

A fast interrupt function must have a `void` return type and cannot have any parameters.

Example

```
__fast_interrupt void my_interrupt_handler(void);
```

See also

*Interrupt functions*, page 23, *vector*, page 198, *INTVEC*, page 235.

## **\_\_interrupt**

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 173.

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.

To override the non-maskable interrupts, use the template file `fixedint.c` in the `src\lib` directory. Non-maskable interrupts do not have a vector number and do not use the `#pragma vector` directive.

Example

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

See also

*Interrupt functions*, page 23, *vector*, page 198, *INTVEC*, page 235.

## **\_\_intrinsic**

Description

The `__intrinsic` keyword is reserved for compiler internal use only.

## **\_\_monitor**

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 176.

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 24. Read also about the intrinsic functions `__disable_interrupt`, page 201, `__enable_interrupt`, page 201, `__get_interrupt_state`, page 203, and `__set_interrupt_state`, page 207.

## **`__no_init`**

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 176.

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 *Object attributes*, page 176.

Description The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

Example `__noreturn void terminate(void);`

## **`__packed`**

Syntax Follows the generic syntax rules for type attributes that can be used on data, see *Type attributes*, page 173.

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

- When used with a `struct` or `union` type definition, the maximum alignment of members of that `struct` or `union` is set to 1, to eliminate any gaps between the members. The type of each members also receives the `__packed` type attribute.

- When used with any other type, the resulting type is the same as the type without the `__packed` type attribute, but with an alignment of 1. Types that already have an alignment of 1 are not affected by the `__packed` type attribute.

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

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

**Example**

```
__packed struct X {char ch; int i;}; /* No pad bytes */
void foo (struct X * xp) /* No need for __packed here */
{
 int * p1 = &xp->i; /* Error:"int *">"int __packed *" */
 int __packed * p2 = &xp->i; /* OK */
 char * p2 = &xp->ch; /* OK, char not affected */
}
```

**See also**

*pack*, page 194.

## **\_\_root**

**Syntax**

Follows the generic syntax rules for object attributes, see *Object attributes*, page 176.

**Description**

A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.

**Example**

```
__root int myarray[10];
```

**See also**

To read more about modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide*.

## **\_\_sdata16**

**Syntax**

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 173.

**Description**

The `__sdata16` memory attribute overrides the default storage of variables and constants given by the selected data model and code model, respectively, and places individual variables and constants in SB-relative memory.

**Storage information**

- Address range: SB register + 0x0–0xFFFF (64 Kbytes)

- Maximum object size: 64 Kbytes
- Pointer size: 4 bytes.

Example `__sbdatal6 int x;`

See also *Memory types*, page 13.

## **\_\_sbdatal6**

**Syntax** Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 173.

**Description** The `__sbdatal6` memory attribute overrides the default storage of variables and constants given by the selected data model and code model, respectively, and places individual variables and constants in SB-relative memory.

**Storage information**

- Address range: SB register + 0xFFFFF (16 Mbytes)
- Maximum object size: 16 Mbytes
- Pointer size: 4 bytes

Example `__sbdatal6 int x;`

See also *Memory types*, page 13.

## **\_\_task**

**Syntax** Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 173.

**Description** This keyword allows functions to exit without restoring registers and it is typically used for the `main` function.

By default, functions save the contents of used non-scratch registers (preserved registers) on the stack upon entry, and restore them at exit. Functions declared `__task` do not save any registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers needed by the calling function, you should only use `__task` on functions that do not return.

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);
```

# Pragma directives

This chapter describes the pragma directives of the compiler.

The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

---

## Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

| Pragma directive            | Description                                                                                                |
|-----------------------------|------------------------------------------------------------------------------------------------------------|
| <code>bitfields</code>      | Controls the order of bitfield members                                                                     |
| <code>constseg</code>       | Places constant variables in a named segment                                                               |
| <code>data_alignment</code> | Gives a variable a higher (more strict) alignment                                                          |
| <code>dataseg</code>        | Places variables in a named segment                                                                        |
| <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>include_alias</code>  | Specifies an alias for an include file                                                                     |
| <code>inline</code>         | Inlines 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                                                                                     |
|------------------|-------------------------------------------------------------------------------------------------|
| object_attribute | Changes the definition of a variable or a function                                              |
| optimize         | Specifies the type and level of an optimization                                                 |
| pack             | Specifies the alignment of structures and union members                                         |
| __printf_args    | Verifies that a function with a printf-style format string is called with the correct arguments |
| required         | Ensures that a symbol that is needed by another symbol is included in the linked output         |
| rtmodel          | Adds a runtime model attribute to the module                                                    |
| __scanf_args     | Verifies that a function with a scanf-style format string is called with the correct arguments  |
| segment          | Declares a segment name to be used by intrinsic functions                                       |
| type_attribute   | Changes the declaration and definitions of a variable or function                               |
| vector           | Specifies the vector of an interrupt function                                                   |

Table 35: Pragma directives summary (Continued)

**Note:** For portability reasons, see also *Recognized pragma directives (6.8.6)*, page 245.

## Descriptions of pragma directives

This section gives detailed information about each pragma directive.

### 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.<br><br>By default, the compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the #pragma bitfields=reversed directive to place the bitfield members from the most significant to the least significant |                                                                                         |

bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

See also

*Bitfields*, page 155.

## constseg

Syntax

```
#pragma constseg=[__memoryattribute][SEGMENT_NAME|default]
```

Parameters

|                                |                                                                                                                            |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>__memoryattribute</code> | An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used. |
| <code>SEGMENT_NAME</code>      | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.                       |
| <code>default</code>           | Uses the default segment for constants.                                                                                    |

Description

Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma constseg=__data24 MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

## data\_alignment

Syntax

```
#pragma data_alignment=expression
```

Parameters

|                         |                                                          |
|-------------------------|----------------------------------------------------------|
| <code>expression</code> | 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 than it would otherwise have. It 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.

## dataseg

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                            |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | #pragma dataseg=[__memoryattribute ] {SEGMENT_NAME  default}                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                            |
| Parameters  | <i>__memoryattribute</i>                                                                                                                                                                                                                                                                                                                                                                                                              | An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used. |
|             | <i>SEGMENT_NAME</i>                                                                                                                                                                                                                                                                                                                                                                                                                   | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.                       |
|             | default                                                                                                                                                                                                                                                                                                                                                                                                                               | Uses the default segment.                                                                                                  |
| Description | Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared <code>__no_init</code> . The setting remains active until you turn it off again with the <code>#pragma constseg=default</code> directive. |                                                                                                                            |
| Example     | <pre>#pragma dataseg=__data24 MY_SEGMENT __no_init char myBuffer[1000]; #pragma dataseg=default</pre>                                                                                                                                                                                                                                                                                                                                 |                                                                                                                            |

## diag\_default

|             |                                                                                                                                                                                                                                                                                                                                       |                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Syntax      | #pragma diag_default=tag[, tag, ...]                                                                                                                                                                                                                                                                                                  |                                                                           |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                                                                            | The number of a diagnostic message, for example the message number Pe117. |
|             |                                                                                                                                                                                                                                                                                                                                       |                                                                           |
| Description | Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options <code>--diag_error</code> , <code>--diag_remark</code> , <code>--diag_suppress</code> , or <code>--diag_warnings</code> , for the diagnostic messages specified with the tags. |                                                                           |
| See also    | <i>Diagnostics</i> , page 122.                                                                                                                                                                                                                                                                                                        |                                                                           |



## diag\_error

|             |                                                                                                             |                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Syntax      | #pragma diag_error=tag[, tag, ...]                                                                          |                                                                           |
| Parameters  | <i>tag</i>                                                                                                  | The number of a diagnostic message, for example the message number Pe117. |
| Description | Use this pragma directive to change the severity level to <code>error</code> for the specified diagnostics. |                                                                           |
| See also    | <i>Diagnostics</i> , page 122.                                                                              |                                                                           |

## diag\_remark

|             |                                                                                                                      |                                                                           |
|-------------|----------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Syntax      | #pragma diag_remark=tag[, tag, ...]                                                                                  |                                                                           |
| 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 122.                                                                                       |                                                                           |

## diag\_suppress

|             |                                                                          |                                                                           |
|-------------|--------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Syntax      | #pragma diag_suppress=tag[, tag, ...]                                    |                                                                           |
| 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 122.                                           |                                                                           |

## diag\_warning

|             |                                                                                                          |                                                                           |
|-------------|----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Syntax      | #pragma diag_warning=tag[, tag, ...]                                                                     |                                                                           |
| 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 warning for the specified diagnostic messages. |                                                                           |
| See also    | <i>Diagnostics</i> , page 122.                                                                           |                                                                           |

## include\_alias

|             |                                                                                                                                                                                                                                                                                                                                                        |                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| Syntax      | #pragma include_alias ("orig_header" , "subst_header")<br>#pragma include_alias (<orig_header> , <subst_header>)                                                                                                                                                                                                                                       |                                                                  |
| Parameters  | <i>orig_header</i>                                                                                                                                                                                                                                                                                                                                     | The name of a header file for which you want to create an alias. |
|             | <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 #include directives and subst_header must match its corresponding #include directive exactly. |                                                                  |
| Example     | #pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)<br>#include <stdio.h><br><br>This example will substitute the relative file <code>stdio.h</code> with a counterpart located according to the specified path.                                                                                                                                |                                                                  |
| See also    | <i>Include file search procedure</i> , page 120.                                                                                                                                                                                                                                                                                                       |                                                                  |

## inline

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                         |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------|
| Syntax      | <code>#pragma inline[=forced]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                         |
| Parameters  | <code>forced</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Disables the compiler's heuristics and forces inlining. |
| Description | <p>Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually occurs is subject to the compiler's heuristics.</p> <p>This is similar to the C++ keyword <code>inline</code>, but has the advantage of being available in C code.</p> <p>Specifying <code>#pragma inline=forced</code> disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like <code>printf</code>), an error message is emitted.</p> <p><b>Note:</b> Because specifying <code>#pragma inline=forced</code> disables the compiler's heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.</p> |                                                         |

## language

|             |                                                                                                                                        |                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma language={extended default}</code>                                                                                       |                                                                                                                |
| Parameters  | <code>extended</code>                                                                                                                  | Turns on the IAR Systems language extensions and turns off the <code>--strict_ansi</code> command line option. |
|             | <code>default</code>                                                                                                                   | Uses the language settings specified by compiler options.                                                      |
| Description | Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line. |                                                                                                                |

location

|             |                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                      |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| Syntax      | #pragma location={ <i>address</i>   <i>NAME</i> }                                                                                                                                                                                                                                                                                                                                                           |                                                                                                      |
| Parameters  | <i>address</i>                                                                                                                                                                                                                                                                                                                                                                                              | The absolute address of the global or static variable for which you want an absolute location.       |
|             | <i>NAME</i>                                                                                                                                                                                                                                                                                                                                                                                                 | 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 <code>__no_init</code> or <code>const</code> . Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive. |                                                                                                      |
| Example     | <pre>#pragma location=0x2000 __no_init volatile char PORT1; /* PORT1 is located at address                                 0x2000 */  #pragma location="foo" char PORT1; /* PORT1 is located in segment foo */  /* A better way is to use a corresponding mechanism */ #define FLASH _Pragma("location=\"FLASH\"") ... FLASH int i; /* i is placed in the FLASH segment */</pre>                            |                                                                                                      |
| See also    | <i>Controlling data and function placement in memory</i> , page 103.                                                                                                                                                                                                                                                                                                                                        |                                                                                                      |

message

|             |                                                                                                                  |                                                              |
|-------------|------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|
| Syntax      | #pragma message( <i>message</i> )                                                                                |                                                              |
| Parameters  | <i>message</i>                                                                                                   | The message that you want to direct to <code>stdout</code> . |
|             |                                                                                                                  |                                                              |
| Description | Use this pragma directive to make the compiler print a message to <code>stdout</code> when the file is compiled. |                                                              |
| Example:    | <pre>#ifdef TESTING #pragma message("Testing") #endif</pre>                                                      |                                                              |

object\_attribute

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma object_attribute=object_attribute[,object_attribute,...]</code>                                                                                                                                                                                                                                                                                                                                                        |
| Parameters  | For a list of object attributes that can be used with this pragma directive, see <i>Object attributes</i> , page 176.                                                                                                                                                                                                                                                                                                                |
| 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 <code>#pragma type_attribute</code> that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations. |
| Example     | <pre>#pragma object_attribute=__no_init char bar;</pre>                                                                                                                                                                                                                                                                                                                                                                              |
| See also    | <i>General syntax rules for extended keywords</i> , page 173.                                                                                                                                                                                                                                                                                                                                                                        |

optimize

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                       |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma optimize=param[ param...]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                       |
| Parameters  | <code>balanced size speed</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Optimizes balanced between speed and size, optimizes for size, or optimizes for speed |
|             | <code>none low medium high</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | Specifies the level of optimization                                                   |
|             | <code>no_code_motion</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | Turns off code motion                                                                 |
|             | <code>no_cse</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Turns off common subexpression elimination                                            |
|             | <code>no_inline</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | Turns off function inlining                                                           |
|             | <code>no_tbaa</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | Turns off type-based alias analysis                                                   |
|             | <code>no_unroll</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | Turns off loop unrolling                                                              |
| Description | <p>Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.</p> <p>The parameters <code>speed</code>, <code>size</code>, and <code>balanced</code> only have effect on the <code>high</code> optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.</p> |                                                                                       |

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

Example

```
#pragma optimize=speed
int small_and_used_often()
{
 ...
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
 ...
}
```

pack

Syntax

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

Parameters

|             |                                                                      |
|-------------|----------------------------------------------------------------------|
| <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 end of file.

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

See also

*Structure types*, page 158.

## \_\_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 *,...);  /* Function call */ printf("%d",x); /* Compiler checks that x is a double */</pre>                                                                              |

## required

|             |                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma required=symbol</code>                                                                                                                                                                                                                                                                                                                                                              |
| Parameters  | <p><i>symbol</i>                      Any statically linked function or variable.</p>                                                                                                                                                                                                                                                                                                             |
| 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() {...}</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      | #pragma rtmodel="key", "value"                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                      |
| Parameters  | "key"                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | A text string that specifies the runtime model attribute.                                                                                            |
|             | "value"                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all. |
| Description | <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 *. 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 67.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                      |

## \_\_scanf\_args

|             |                                                                                                                                                                                                                                 |  |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | #pragma __scanf_args                                                                                                                                                                                                            |  |
| Description | <p>Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct.</p> |  |
| Example     | <pre>#pragma __scanf_args int printf(char const *,...);  /* Function call */ scanf("%d",x); /* Compiler checks that x is a double */</pre>                                                                                      |  |



## segment

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                              |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma segment="NAME" [__memoryattribute] [align]</code>                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                              |
| 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 part. The value must be a constant integer expression to the power of two.            |
| Description | <p>Use this pragma directive to define a segment name that can be used by the segment operators <code>__segment_begin</code> and <code>__segment_end</code>. All segment declarations for a specific segment must have the same memory type attribute and alignment.</p> <p>If an optional memory attribute is used, the return type of the segment operators <code>__segment_begin</code> and <code>__segment_end</code> is:</p> <pre>void __memoryattribute *.</pre> |                                                                                                                              |
| Example     | <code>#pragma segment="MYHUGE" __data32 4</code>                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                              |
| See also    | <i>Important language extensions</i> , page 164. For more information about segments and segment parts, see the chapter <i>Placing code and data</i> .                                                                                                                                                                                                                                                                                                                 |                                                                                                                              |

## type\_attribute

|             |                                                                                                                                                                                                                                                                                                                                                                                                  |  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>#pragma type_attribute=type_attribute[, type_attribute, ...]</code>                                                                                                                                                                                                                                                                                                                        |  |
| Parameters  | For a list of type attributes that can be used with this pragma directive, see <i>Type attributes</i> , page 173.                                                                                                                                                                                                                                                                                |  |
| Description | <p>Use this pragma directive to specify IAR-specific <i>type attributes</i>, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute might not be applicable to all kind of objects.</p> <p>This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.</p> |  |
| Example     | <p>In this example, an <code>int</code> object with the memory attribute <code>__data16</code> is defined:</p> <pre>#pragma type_attribute=__data16 int x;</pre>                                                                                                                                                                                                                                 |  |

This declaration, which uses extended keywords, is equivalent:

```
__data16 int x;
```

See also                      See the chapter *Extended keywords* for more details.

**vector**

Syntax                      `#pragma vector=vector1[, vector2, vector3, ...]`

Parameters

|               |                                                |
|---------------|------------------------------------------------|
| <i>vector</i> | The vector number(s) of an interrupt function. |
|---------------|------------------------------------------------|

Description

Use this pragma directive to specify the vector(s) of an interrupt function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example!

```
#pragma vector=0x14
__interrupt void my_handler(void);
```

# 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>__break</code>               | Inserts a BRK instruction             |
| <code>__delay_cycles</code>        | Inserts code to delay execution       |
| <code>__disable_interrupt</code>   | Disables interrupts                   |
| <code>__enable_interrupt</code>    | Enables interrupts                    |
| <code>__exchange_byte</code>       | Inserts an XCHG instruction           |
| <code>__exchange_long</code>       | Inserts an XCHG instruction           |
| <code>__exchange_word</code>       | Inserts an XCHG instruction           |
| <code>__get_DCR_register</code>    | Returns the value of the DCR register |
| <code>__get_DCT_register</code>    | Returns the value of the DCT register |
| <code>__get_DDA_register</code>    | Returns the value of the DDA register |
| <code>__get_DDR_register</code>    | Returns the value of the DDR register |
| <code>__get_DMD_register</code>    | Returns the value of the DMD register |
| <code>__get_DSA_register</code>    | Returns the value of the DSA register |
| <code>__get_DSR_register</code>    | Returns the value of the DSR register |
| <code>__get_interrupt_level</code> | Returns the interrupt level           |

*Table 36: Intrinsic functions summary*

| Intrinsic function                   | Description                                  |
|--------------------------------------|----------------------------------------------|
| <code>__get_interrupt_state</code>   | Returns the interrupt state                  |
| <code>__get_interrupt_table</code>   | Returns the value of the INTB register       |
| <code>__get_VCT_register</code>      | Returns the value of the VCT register        |
| <code>__illegal_opcode</code>        | Inserts an UND instruction                   |
| <code>__interrupt_on_overflow</code> | Inserts an INTO instruction                  |
| <code>__load_context</code>          | Inserts an LDCTX instruction                 |
| <code>__low_level_init</code>        | Low-level initialization                     |
| <code>__no_operation</code>          | Inserts a NOP instruction                    |
| <code>__RMPA_B</code>                | Inserts an RMPA instruction                  |
| <code>__RMPA_L</code>                | Inserts an RMPA instruction                  |
| <code>__RMPA_W</code>                | Inserts an RMPA instruction                  |
| <code>__ROUND</code>                 | Inserts a ROUND instruction                  |
| <code>__set_DCR_register</code>      | Writes a specific value to the DCR register  |
| <code>__set_DCT_register</code>      | Writes a specific value to the DCT register  |
| <code>__set_DDA_register</code>      | Writes a specific value to the DDA register  |
| <code>__set_DDR_register</code>      | Writes a specific value to the DDR register  |
| <code>__set_DMD_register</code>      | Writes a specific value to the DMD register  |
| <code>__set_DSA_register</code>      | Writes a specific value to the DSA register  |
| <code>__set_DSR_register</code>      | Writes a specific value to the DSR register  |
| <code>__set_interrupt_level</code>   | Sets the interrupt level                     |
| <code>__set_interrupt_state</code>   | Restores the interrupt state                 |
| <code>__set_interrupt_table</code>   | Writes a specific value to the INTB register |
| <code>__set_VCT_register</code>      | Writes a specific value to the VCT register  |
| <code>__SIN</code>                   | Inserts an SIN instruction                   |
| <code>__software_interrupt</code>    | Inserts an INT instruction                   |
| <code>__SOUT</code>                  | Inserts an SOUT instruction                  |
| <code>__STOP</code>                  | Inserts a STOP instruction                   |
| <code>__store_context</code>         | Inserts an STCTX instruction                 |
| <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.

### **\_\_break**

Syntax `void __break(void);`

Description Inserts a BRK instruction.

### **\_\_delay\_cycles**

Syntax `void __delay_cycles(unsigned long cycles);`

Description Inserts code to delay execution for *cycles* number of execution cycles.

### **\_\_disable\_interrupt**

Syntax `void __disable_interrupt(void);`

Description Disables interrupts by clearing the I bit in the status register FLG.

### **\_\_enable\_interrupt**

Syntax `void __enable_interrupt(void);`

Description Enables interrupts by setting the I bit in the status register FLG.

### **\_\_exchange\_byte**

Syntax `unsigned char __exchange_byte(unsigned char src,  
unsigned char * dst);`

Description Inserts an XCHG.B *src,dst* instruction.

## **\_\_exchange\_long**

|             |                                                                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __exchange_long(unsigned long <i>src</i>,<br/>                                  unsigned long * <i>dst</i>);</code> |
| Description | Inserts an <code>XCHG.L <i>src</i>,<i>dst</i></code> instruction.                                                                       |

## **\_\_exchange\_word**

|             |                                                                                                                                            |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned short __exchange_word(unsigned short <i>src</i>,<br/>                                  unsigned short * <i>dst</i>);</code> |
| Description | Inserts an <code>XCHG.W <i>src</i>,<i>dst</i></code> instruction.                                                                          |

## **\_\_get\_DCR\_register**

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __get_DCR_register(unsigned char <i>n</i>);</code>                                                   |
| Description | Returns the value of the <code>DCR<sub><i>n</i></sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

## **\_\_get\_DCT\_register**

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __get_DCT_register(unsigned char <i>n</i>);</code>                                                   |
| Description | Returns the value of the <code>DCT<sub><i>n</i></sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

## **\_\_get\_DDA\_register**

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __get_DDA_register(unsigned char <i>n</i>);</code>                                                   |
| Description | Returns the value of the <code>DDA<sub><i>n</i></sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

## **\_\_get\_DDR\_register**

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __get_DDR_register(unsigned char <i>n</i>);</code>                                                   |
| Description | Returns the value of the <code>DDR<sub><i>n</i></sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

**\_\_get\_DMD\_register**

|             |                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __get_DMD_register(unsigned char n);</code>                                                   |
| Description | Returns the value of the <code>DMD<sub>n</sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

**\_\_get\_DSA\_register**

|             |                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __get_DSA_register(unsigned char n);</code>                                                   |
| Description | Returns the value of the <code>DSA<sub>n</sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

**\_\_get\_DSR\_register**

|             |                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned long __get_DSR_register(unsigned char n);</code>                                                   |
| Description | Returns the value of the <code>DSR<sub>n</sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

**\_\_get\_interrupt\_level**

|             |                                                                                                                                                                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>__ilevel_t __get_interrupt_level(void);</code>                                                                                                                                                                                                                                                                  |
| Description | <p>Returns the current interrupt level. The return type <code>__ilevel_t</code> has the following definition:</p> <pre>typedef unsigned char __ilevel_t;</pre> <p>The return value of <code>__get_interrupt_level</code> can be used as an argument to the <code>__set_interrupt_level</code> intrinsic function.</p> |

**\_\_get\_interrupt\_state**

|             |                                                                                                                                                                                       |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>__istate_t __get_interrupt_state(void);</code>                                                                                                                                  |
| Description | Returns the global interrupt state. The return value can be used as an argument to the <code>__set_interrupt_state</code> intrinsic function, which will restore the interrupt state. |

Example

```
__istate_t s = __get_interrupt_state();
__disable_interrupt();

/* Do something */

__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.

## **\_\_get\_interrupt\_table**

Syntax

```
unsigned long __get_interrupt_table(void);
```

Description

Returns the value of the `INTB` register.

## **\_\_get\_VCT\_register**

Syntax

```
unsigned long __get_VCT_register(void);
```

Description

Returns the value of the `VCT` register.

## **\_\_illegal\_opcode**

Syntax

```
void __illegal_opcode(void);
```

Description

Inserts an `UND` instruction.

## **\_\_interrupt\_on\_overflow**

Syntax

```
void __interrupt_on_overflow(void);
```

Description

Inserts an `INTO` instruction.

## **\_\_load\_context**

Syntax

```
void __load_context(void __data16 * src, void __data24 * dst);
```

Description

Inserts an `LDCTX src, dst` instruction.



**\_\_low\_level\_init**

|             |                                                                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int __low_level_init(void);</code>                                                                                                                                                  |
| Description | Gives the application a chance to perform early initializations. The function is called from the file <code>cstartup.s53</code> . See <i>Customizing system initialization</i> , page 55. |

**\_\_no\_operation**

|             |                                         |
|-------------|-----------------------------------------|
| Syntax      | <code>void __no_operation(void);</code> |
| Description | Inserts a NOP instruction.              |

**\_\_RMPA\_B**

|             |                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>long long __RMPA_B(signed char * v1, signed char * v2, unsigned long n, long long acc);</code>                                                                                                                                                                                                                                                                      |
| Description | Inserts an <code>RMPA.B</code> instruction. The <code>RMPA</code> instruction sequentially multiplies the two vectors <code>v1</code> and <code>v2</code> and adds each product to the accumulator <code>acc</code> . The length of the vectors is <code>n</code> . You can supply an initial value for the accumulator <code>acc</code> , either variable or a constant. |

**\_\_RMPA\_L**

|             |                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>long long __RMPA_L(signed long * v1, signed long * v2, unsigned long n, long long acc);</code>                                                                                                                                                                                                                                                                      |
| Description | Inserts an <code>RMPA.L</code> instruction. The <code>RMPA</code> instruction sequentially multiplies the two vectors <code>v1</code> and <code>v2</code> and adds each product to the accumulator <code>acc</code> . The length of the vectors is <code>n</code> . You can supply an initial value for the accumulator <code>acc</code> , either variable or a constant. |

**\_\_RMPA\_W**

|             |                                                                                                                                                                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>long long __RMPA_W(signed short * v1, signed short * v2, unsigned long n, long long acc);</code>                                                                                                                                  |
| Description | Inserts an <code>RMPA.W</code> instruction. The <code>RMPA</code> instruction sequentially multiplies the two vectors <code>v1</code> and <code>v2</code> and adds each product to the accumulator <code>acc</code> . The length of the |

vectors is  $n$ . You can supply an initial value for the accumulator  $acc$ , either variable or a constant.

## **\_\_ROUND**

Syntax

```
int __ROUND(float);
```

Description

Inserts a `ROUND` instruction. See *Casting a floating-point value to an integer*, page 101.

## **\_\_set\_DCR\_register**

Syntax

```
void __set_DCR_register(unsigned char n, unsigned long value);
```

Description

Writes a specific value to the  $DCR_n$  register. The channel  $n$  can be either 0, 1, 2, or 3.

## **\_\_set\_DCT\_register**

Syntax

```
void __set_DCT_register(unsigned char n, unsigned long value);
```

Description

Writes a specific value to the  $DCT_n$  register. The channel  $n$  can be either 0, 1, 2, or 3.

## **\_\_set\_DDA\_register**

Syntax

```
void __set_DDA_register(unsigned char n, unsigned long value);
```

Description

Writes a specific value to the  $DDA_n$  register. The channel  $n$  can be either 0, 1, 2, or 3.

## **\_\_set\_DDR\_register**

Syntax

```
void __set_DDR_register(unsigned char n, unsigned long value);
```

Description

Writes a specific value to the  $DDR_n$  register. The channel  $n$  can be either 0, 1, 2, or 3.

## **\_\_set\_DMD\_register**

Syntax

```
void __set_DMD_register(unsigned char n, unsigned long value);
```

Description

Writes a specific value to the  $DMD_n$  register. The channel  $n$  can be either 0, 1, 2, or 3.

**\_\_set\_DSA\_register**

|             |                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __set_DSA_register(unsigned char <i>n</i>, unsigned long <i>value</i>);</code>                                      |
| Description | Writes a specific value to the <code>DSA<sub><i>n</i></sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

**\_\_set\_DSR\_register**

|             |                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __set_DSR_register(unsigned char <i>n</i>, unsigned long <i>value</i>);</code>                                      |
| Description | Writes a specific value to the <code>DSR<sub><i>n</i></sub></code> register. The channel <i>n</i> can be either 0, 1, 2, or 3. |

**\_\_set\_interrupt\_level**

|             |                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __set_interrupt_level(__ilevel_t);</code>                                                                                                 |
| Description | Sets the interrupt level. For information about the <code>__ilevel_t</code> type, see <a href="#"><code>__get_interrupt_level</code></a> , page 203. |

**\_\_set\_interrupt\_state**

|              |                                                                                                                                                                                                                                                  |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>void __set_interrupt_state(__istate_t);</code>                                                                                                                                                                                             |
| Descriptions | Restores the interrupt state by setting the value returned by the <code>__get_interrupt_state</code> function.<br><br>For information about the <code>__istate_t</code> type, see <a href="#"><code>__get_interrupt_state</code></a> , page 203. |

**\_\_set\_interrupt\_table**

|             |                                                                        |
|-------------|------------------------------------------------------------------------|
| Syntax      | <code>void __set_interrupt_table(unsigned long <i>address</i>);</code> |
| Description | Writes a specific value to the <code>INTB</code> register.             |

**\_\_set\_VCT\_register**

|             |                                                                     |
|-------------|---------------------------------------------------------------------|
| Syntax      | <code>void __set_VCT_register(unsigned long <i>address</i>);</code> |
| Description | Writes a specific value to the <code>VCT</code> register.           |

## **\_\_SIN**

Syntax `void __SIN(char * src, char * dst, unsigned long n);`

Description Inserts an `SIN.B` instruction.

*src*           The source register address  
*dst*           The destination char array pointer  
*n*             The number of bytes

## **\_\_software\_interrupt**

Syntax `void __software_interrupt(unsigned char n);`

Description Inserts an `INT #n` instruction.

## **\_\_SOUT**

Syntax `void __SOUT(char * src, char * dst, unsigned long n);`

Description Inserts an `SOUT.B` instruction.

*src*           The source char array pointer  
*dst*           The destination register address  
*n*             The number of bytes

## **\_\_STOP**

Syntax `void __STOP(void);`

Description Inserts a `STOP` instruction.

## **\_\_store\_context**

Syntax `void __store_context(void __data16 * src, void __data24 * dst);`

Description Inserts an `STCTX src, dst` instruction.

**\_\_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 R32C adheres to the ISO/ANSI standard. The compiler also makes these preprocessor-related features available to you:

- **Predefined preprocessor symbols**  
These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 212.
- **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 131.
- **Preprocessor extensions**  
There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 214.
- **Preprocessor output**  
Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 149.

Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 244.

# Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

| Predefined symbol    | Identifies                                                                                                                                                                                                                                                                                                                                  |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| __BASE_FILE__        | A string that identifies the name of the base source file (that is, not the header file), being compiled. See also __FILE__, page 212, and <code>--no_path_in_file_macros</code> , page 144.                                                                                                                                                |
| __BUILD_NUMBER__     | 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__       | An integer that identifies the code model in use. The symbol reflects the <code>--code_model</code> option and is defined to __FAR__ or __HUGE__. These symbolic names can be used when testing the __CODE_MODEL__ symbol.                                                                                                                  |
| __cplusplus          | An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.* |
| __DATA_MODEL__       | An integer that identifies the data model in use. The symbol reflects the <code>--data_model</code> option and can be defined to __NEAR__, __FAR__, or __HUGE__. These symbolic names can be used when testing the __DATA_MODEL__ symbol.                                                                                                   |
| __DATE__             | A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Oct 30 2005".*                                                                                                                                                                                                                  |
| __embedded_cplusplus | 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 <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*                        |
| __FILE__             | A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also __BASE_FILE__, page 212, and <code>--no_path_in_file_macros</code> , page 144.*                                                                                                                 |

Table 37: Predefined symbols



| Predefined symbol                | Identifies                                                                                                                                                                                                                                                                                                                                                |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__func__</code>            | A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 137. See also <code>__PRETTY_FUNCTION__</code> , page 213.                                                                                 |
| <code>__FUNCTION__</code>        | A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 137. See also <code>__PRETTY_FUNCTION__</code> , page 213.                                                                                 |
| <code>__IAR_SYSTEMS_ICC__</code> | An integer that identifies the IAR compiler platform. The current value is 7. Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was compiled by a compiler from IAR Systems.                                                                         |
| <code>__ICC32C__</code>          | An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for R32C, and otherwise to 0.                                                                                                                                                                                                                                           |
| <code>__LINE__</code>            | An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.*                                                                                                                                                                                               |
| <code>__LITTLE_ENDIAN__</code>   | An integer that identifies the byte order of the microcomputer. For the R32C/100 microcomputer families, the value of this symbol is defined to 1 (TRUE), which means that the byte order is little-endian.                                                                                                                                               |
| <code>__PRETTY_FUNCTION__</code> | A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example "void func(char)". This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 137. See also <code>__func__</code> , page 213. |
| <code>__STDC__</code>            | An integer that is set to 1, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with <code>#ifdef</code> to detect whether the compiler in use adheres to ISO/ANSI C.*                                                                                                                                                |
| <code>__STDC_VERSION__</code>    | An integer that identifies the version of ISO/ANSI C standard in use. The symbol expands to 199409L. This symbol does not apply in EC++ mode.*                                                                                                                                                                                                            |

Table 37: Predefined symbols (Continued)

| Predefined symbol | Identifies                                                                                                                                                                                                                                                                                                                              |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| __SUBVERSION__    | An integer that identifies the version letter of the compiler version number, for example the C in 4.21C, as an ASCII character.                                                                                                                                                                                                        |
| __TIME__          | A string that identifies the time of compilation in the form "hh:mm:ss".*                                                                                                                                                                                                                                                               |
| __VER__           | An integer that identifies the version number of the IAR compiler in use. 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. |

Table 37: Predefined symbols (Continued)

\* This symbol is required by the ISO/ANSI standard.

## Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI directives.

### NDEBUG

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.



In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

## **`_Pragma()`**

|             |                                                                                                                                                                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre><code>_Pragma("string")</code></pre> <p>where <i>string</i> follows the syntax of the corresponding pragma directive.</p>                                                                                                                                           |
| Description | <p>This preprocessor operator is part of the C99 standard and can be used, for example, in defines and is equivalent to the <code>#pragma</code> directive.</p> <p><b>Note:</b> The <code>-e</code> option—enable language extensions—does not have to be specified.</p> |
| Example     | <pre><code>#if NO_OPTIMIZE     #define NOOPT _Pragma("optimize=none") #else     #define NOOPT #endif</code></pre>                                                                                                                                                        |
| See also    | See the chapter <i>Pragma directives</i> .                                                                                                                                                                                                                               |

## **`#warning message`**

|             |                                                                                                                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre><code>#warning message</code></pre> <p>where <i>message</i> can be any string.</p>                                                                                                                  |
| Description | Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard <code>#error</code> directive is used. |

## **`__VA_ARGS__`**

|             |                                                                                                                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre><code>#define P(...)    __VA_ARGS__ #define P(x,y,...)  x + y + __VA_ARGS__</code></pre> <p><code>__VA_ARGS__</code> will contain all variadic arguments concatenated, including the separating commas.</p> |
| Description | Variadic macros are the preprocessor macro equivalents of <code>printf</code> style functions. <code>__VA_ARGS__</code> is part of the C99 standard.                                                             |

### Example

```
#if DEBUG
 #define DEBUG_TRACE(S,...) printf(S,__VA_ARGS__)
#else
 #define DEBUG_TRACE(S,...)
#endif
/* Place your own code here */
DEBUG_TRACE("The value is:%d\n",value);

will result in:

printf("The value is:%d\n",value);
```

# Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

---

## Introduction

The compiler is delivered with the IAR DLIB Library; a complete ISO/ANSI C and C++ library. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

## HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

## LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic settings for project configuration*, page 5. The linker will include only those routines that are required—directly or indirectly—by your application.

## REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but these functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- The miscellaneous functions `setlocale`, `rand`, `atexit`, `strerror`, `strtok`
- Functions that use files in some way. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

Some functions also share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. Among these functions are:

`exp`, `exp10`, `ldexp`, `log`, `log10`, `pow`, `sqrt`, `acos`, `asin`, `atan2`,  
`cosh`, `sinh`, `strtod`, `strtol`, `strtoul`

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

---

## IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.

- Intrinsic functions, allowing low-level use of R32C/100 features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, partly taken from the C99 standard, see *Added C functionality*, page 222.

## C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Compiler extensions*.

The following table lists the C header files:

| Header file             | Usage                                                              |
|-------------------------|--------------------------------------------------------------------|
| <code>assert.h</code>   | Enforcing assertions when functions execute                        |
| <code>ctype.h</code>    | Classifying characters                                             |
| <code>errno.h</code>    | Testing error codes reported by library functions                  |
| <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>time.h</code>     | Converting between various time and date formats                   |
| <code>wchar.h</code>    | Support for wide characters                                        |
| <code>wctype.h</code>   | Classifying wide characters                                        |

Table 38: Traditional standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

| Header file | Usage                                                                             |
|-------------|-----------------------------------------------------------------------------------|
| complex     | Defining a class that supports complex arithmetic                                 |
| exception   | Defining several functions that control exception handling                        |
| fstream     | Defining several I/O stream classes that manipulate external files                |
| iomanip     | Declaring several I/O stream manipulators that take an argument                   |
| ios         | Defining the class that serves as the base for many I/O streams classes           |
| iosfwd      | Declaring several I/O stream classes before they are necessarily defined          |
| iostream    | Declaring the I/O stream objects that manipulate the standard streams             |
| istream     | Defining the class that performs extractions                                      |
| new         | Declaring several functions that allocate and free storage                        |
| ostream     | Defining the class that performs insertions                                       |
| sstream     | Defining several I/O stream classes that manipulate string containers             |
| stdexcept   | Defining several classes useful for reporting exceptions                          |
| streambuf   | Defining classes that buffer I/O stream operations                                |
| string      | Defining a class that implements a string container                               |
| strstream   | Defining several I/O stream classes that manipulate in-memory character sequences |

Table 39: Embedded C++ header files

The following table lists additional C++ header files:

| Header file | Usage                                                                 |
|-------------|-----------------------------------------------------------------------|
| fstream.h   | Defining several I/O stream classes that manipulate external files    |
| iomanip.h   | Declaring several I/O stream manipulators that take an argument       |
| iostream.h  | Declaring the I/O stream objects that manipulate the standard streams |
| new.h       | Declaring several functions that allocate and free storage            |

Table 40: Additional Embedded C++ header files—DLIB



### Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

| 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   |
| <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 41: Standard template library header files

### Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

| 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> |

Table 42: New standard C header files—DLIB

| Header file          | Usage                                                  |
|----------------------|--------------------------------------------------------|
| <code>climits</code> | Testing integer type properties                        |
| <code>clocale</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>stdbool</code> | Adds support for the <code>bool</code> data type in C. |
| <code>stddef</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>string</code>  | Manipulating several kinds of strings                  |
| <code>ctime</code>   | Converting between various time and date formats       |
| <code>wchar</code>   | Support for wide characters                            |
| <code>wctype</code>  | Classifying wide characters                            |

Table 42: 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, partly taken from the C99 standard.

The following include files provide these features:

- `ctype.h`
- `inttypes.h`
- `math.h`
- `stdbool.h`
- `stdint.h`
- `stdio.h`
- `stdlib.h`
- `wchar.h`

- `wctype.h`

## **`ctype.h`**

In `ctype.h`, the C99 function `isblank` is defined.

## **`inttypes.h`**

This include file defines the formatters for all types defined in `stdint.h` to be used by the functions `printf`, `scanf`, and all their variants.

## **`math.h`**

In `math.h` all functions exist in a `float` variant and a `long double` variant, suffixed by `f` and `l` respectively. For example, `sinf` and `sinl`.

The following C99 macro symbols are defined:

`HUGE_VALF`, `HUGE_VALL`, `INFINITY`, `NAN`, `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`, `MATH_ERRNO`, `MATH_ERREXCEPT`, `math_errhandling`.

The following C99 macro functions are defined:

`fpclassify`, `signbit`, `isfinite`, `isinf`, `isnan`, `isnormal`, `isgreater`, `isless`, `islessequal`, `islessgreater`, `isunordered`.

The following C99 type definitions are added:

`float_t`, `double_t`.

## **`stdbool.h`**

This include file makes the `bool` type available if the **Allow IAR extensions** (`-e`) option is used.

## **`stdint.h`**

This include file provides integer characteristics.

## **`stdio.h`**

In `stdio.h`, the following C99 functions are defined:

`vscanf`, `vfscanf`, `vsscanf`, `vsnprintf`, `snprintf`

The functions `printf`, `scanf`, and all their variants have added functionality from the C99 standard. For reference information about these functions, see the library reference available from the **Help** menu.

The following functions providing I/O functionality for libraries built without `FILE` support are defined:

`__write_array` Corresponds to `fwrite` on `stdout`.

`__ungetchar` Corresponds to `ungetc` on `stdout`.

`__gets` Corresponds to `fgets` on `stdin`.

## **stdlib.h**

In `stdlib.h`, the following C99 functions are defined:

`_Exit`, `llabs`, `lldiv`, `strtoll`, `strtoull`, `atoll`, `strtof`, `strtold`.

The function `strtod` has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The `__qsortbbl` function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

## **wchar.h**

In `wchar.h`, the following C99 functions are defined:

`vfwscanf`, `vswscanf`, `wscanf`, `wcstof`, `wcstolb`.

## **wctype.h**

In `wctype.h`, the C99 function `iswblank` is defined.

# 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                                                                                           |
|-----------|-------------------------------------------------------------------------------------------------------|
| CHECKSUM  | Holds the checksum generated by the linker.                                                           |
| CODE24    | Holds <code>__code24</code> program code.                                                             |
| CODE32    | Holds <code>__code32</code> program code.                                                             |
| CSTACK    | Holds the stack used by C or C++ programs.                                                            |
| CSTART    | Holds the startup code.                                                                               |
| DATA16_AC | Holds <code>__data16</code> located constant data.                                                    |
| DATA16_AN | Holds <code>__data16</code> located uninitialized data.                                               |
| DATA16_C  | Holds <code>__data16</code> constant data.                                                            |
| 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.                             |
| DATA24_AC | Holds <code>__data24</code> located constant data.                                                    |
| DATA24_AN | Holds <code>__data24</code> located uninitialized data.                                               |
| DATA24_C  | Holds <code>__data24</code> constant data.                                                            |
| 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.                                    |

*Table 43: Segment summary*

| Segment     | Description                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------|
| DATA24_Z    | Holds zero-initialized __data24 static and global variables.                                                                  |
| DATA32_AC   | Holds __data32 located constant data.                                                                                         |
| DATA32_AN   | Holds __data32 located uninitialized data.                                                                                    |
| DATA32_C    | Holds __data32 constant data.                                                                                                 |
| DATA32_I    | Holds __data32 static and global initialized variables.                                                                       |
| DATA32_ID   | Holds initial values for __data32 static and global variables in DATA32_I.                                                    |
| DATA32_N    | Holds __no_init __data32 static and global variables.                                                                         |
| DATA32_Z    | Holds zero-initialized __data32 static and global variables.                                                                  |
| DIFUNCT     | Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before main is called. |
| HEAP        | Holds the heap data used by malloc and free.                                                                                  |
| INTVEC      | Contains all interrupt vectors except for non-maskable interrupts                                                             |
| ISTACK      | Holds the interrupt stack used by C or C++ programs.                                                                          |
| NMIVEC      | Contains the reset and non-maskable interrupt vectors.                                                                        |
| SBDATA16_I  | Holds __sbddata16 static and global initialized variables.                                                                    |
| SBDATA16_ID | Holds initial values for __sbddata16 static and global variables in SBDATA16_I.                                               |
| SBDATA16_N  | Holds __no_init __sbddata16 static and global variables.                                                                      |
| SBDATA16_Z  | Holds zero-initialized __sbddata16 static and global variables.                                                               |
| SBDATA24_I  | Holds __sbddata24 static and global initialized variables.                                                                    |
| SBDATA24_ID | Holds initial values for __sbddata24 static and global variables in SBDATA24_I.                                               |
| SBDATA24_N  | Holds __no_init __sbddata24 static and global variables.                                                                      |
| SBDATA24_Z  | Holds zero-initialized __sbddata24 static and global variables.                                                               |

Table 43: 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 8, *XLINK segment memory types*, page 28.

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 command file, see *Customizing the linker command file*, page 28.

For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

CHECKSUM

|                     |                                                    |
|---------------------|----------------------------------------------------|
| Description         | Holds the checksum generated by the linker.        |
| Segment memory type | CONST                                              |
| Memory placement    | This segment can be placed anywhere in ROM memory. |
| Access type         | Read-only                                          |

CODE24

|                     |                                                                |
|---------------------|----------------------------------------------------------------|
| Description         | Holds <code>__code24</code> program code.                      |
| Segment memory type | CODE                                                           |
| Memory placement    | This segment must be placed in the highest 8 Mbytes of memory. |
| Access type         | Read-only                                                      |

CODE32

|                     |                                                |
|---------------------|------------------------------------------------|
| Description         | Holds <code>__code32</code> program code.      |
| 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, the user stack referred to by the <code>USP</code> stack pointer. |
| Segment memory type | DATA                                                                                             |
| Memory placement    | This segment can be placed anywhere in RAM memory.                                               |
| Access type         | Read/write                                                                                       |
| See also            | <i>The internal data stack</i> , page 34.                                                        |

## CSTART

|                     |                                                                                                                                                                                                                                                                                                   |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds the 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 command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CODE                                                                                                                                                                                                                                                                                              |
| Memory placement    | This segment must be placed in the highest 8 Mbytes of memory.                                                                                                                                                                                                                                    |
| Access type         | Read-only                                                                                                                                                                                                                                                                                         |

## DATA16\_AC

|             |                                                                                                                                                                                                                                                                                                                                                 |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__data16</code> located constant data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p> |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA16\_AN

|             |                                                                                                                                                                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__no_init __data16</code> located data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p> |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



## DATA16\_C

|                     |                                                                     |
|---------------------|---------------------------------------------------------------------|
| Description         | Holds <code>__data16</code> constant data.                          |
| Segment memory type | CONST                                                               |
| Memory placement    | This segment must be placed in the highest 32 Kbytes of ROM memory. |
| Access type         | Read-only                                                           |

## DATA16\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__data16</code> static and global initialized variables initialized by copying from the segment <code>DATA16_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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | This segment must be placed in the lowest 32 Kbytes of RAM 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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Memory placement    | This segment can be placed anywhere in ROM 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 lowest 32 Kbytes of RAM memory. |
| Access type         | Read/write                                                         |

**DATA16\_Z**

|                     |                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__data16</code> static and global variables.</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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                |
| Memory placement    | This segment must be placed in the lowest 32 Kbytes of RAM memory.                                                                                                                                                                                                                                                                                  |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                          |

**DATA24\_AC**

|             |                                                                                                                                                                                                                                                                                                                                                 |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__data24</code> located constant data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p> |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**DATA24\_AN**

|             |                                                                                                                                                                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__no_init __data24</code> located data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p> |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA24\_C

|                     |                                                                              |
|---------------------|------------------------------------------------------------------------------|
| Description         | Holds <code>__data24</code> constant data.                                   |
| Segment memory type | CONST                                                                        |
| Memory placement    | This segment must be placed in the lowest or highest 8 Mbytes of ROM memory. |
| Access type         | Read-only                                                                    |

## DATA24\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__data24</code> static and global initialized variables initialized by copying from the segment <code>DATA24_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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | This segment must be placed in the lowest or highest 8 Mbytes of RAM memory.                                                                                                                                                                                                                                                                                                                                                         |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                           |

## DATA24\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__data24</code> static and global variables in the <code>DATA24_I</code> segment. These values are copied from <code>DATA24_ID</code> to <code>DATA24_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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Memory placement    | This segment can be placed anywhere in ROM memory.                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

**DATA24\_N**

|                     |                                                                              |
|---------------------|------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __data24</code> variables.           |
| Segment memory type | DATA                                                                         |
| Memory placement    | This segment must be placed in the lowest or highest 8 Mbytes of RAM memory. |
| Access type         | Read/write                                                                   |

**DATA24\_Z**

|                     |                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__data24</code> static and global variables.</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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                |
| Memory placement    | This segment must be placed in the lowest or highest 8 Mbytes of RAM memory.                                                                                                                                                                                                                                                                        |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                          |

**DATA32\_AC**

|             |                                                                                                                                                                                                                                                                                                                                                 |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__data32</code> located constant data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p> |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**DATA32\_AN**

|             |                                                                                                                                                                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds <code>__no_init __data32</code> located data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p> |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA32\_C

|                     |                                                    |
|---------------------|----------------------------------------------------|
| Description         | Holds <code>__data32</code> constant data.         |
| Segment memory type | CONST                                              |
| Memory placement    | This segment can be placed anywhere in ROM memory. |
| Access type         | Read-only                                          |

## DATA32\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__data32</code> static and global initialized variables initialized by copying from the segment <code>DATA32_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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | This segment can be placed anywhere in RAM 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 command file, the <code>-z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Memory placement    | This segment can be placed anywhere in ROM 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 RAM memory.                 |
| Access type         | Read/write                                                         |

**DATA32\_Z**

|                     |                                                                                                                                                                                                                                                                                                                                              |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds zero-initialized <code>__data32</code> static and global variables.<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 command file, the <code>-Z</code> directive must be used. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                         |
| Memory placement    | This segment can be placed anywhere in RAM memory.                                                                                                                                                                                                                                                                                           |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                   |

**DIFUNCT**

|                     |                                                      |
|---------------------|------------------------------------------------------|
| Description         | Holds the dynamic initialization vector used by C++. |
| Segment memory type | CONST                                                |
| Memory placement    | This segment can be placed anywhere in memory.       |
| Access type         | Read-only                                            |

**HEAP**

|                     |                                                                                                                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the heap used for dynamically allocated data, in other words data allocated by <code>malloc</code> and <code>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 RAM memory. |
| Access type      | Read/write                                         |
| See also         | <i>The heap</i> , page 35.                         |

## 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 | CONST                                                                                                                                                                 |
| Memory placement    | This segment can be placed anywhere in ROM memory.                                                                                                                    |
| Access type         | Read-only                                                                                                                                                             |

## ISTACK

|                     |                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the interrupt stack, referred to by the <code>ISP</code> stack pointer. It is used for non-maskable interrupts and interrupts 0–127. |
| Segment memory type | DATA                                                                                                                                       |
| Memory placement    | This segment can be placed anywhere in RAM memory.                                                                                         |
| Access type         | Read/write                                                                                                                                 |
| See also            | <i>The internal data stack</i> , page 34.                                                                                                  |

## NMIVEC

|                     |                                                                                     |
|---------------------|-------------------------------------------------------------------------------------|
| Description         | Holds the non-maskable interrupt vector table and the reset vector.                 |
| Segment memory type | CONST                                                                               |
| Memory placement    | This segment must be placed in the memory range <code>0xFFFFFDC–0xFFFFFFFF</code> . |
| Access type         | Read-only                                                                           |

**SBDATA16\_I**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__sbddata16</code> static and global initialized variables initialized by copying from the segment <code>SBDATA16_ID</code> at application startup. Also holds <code>__sbddata16</code> constant data.</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 command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Memory placement    | This segment must be placed within the first 64 Kbytes of memory after the static base reference point ( <code>SBREF</code> ).                                                                                                                                                                                                                                                                                                                                                               |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

**SBDATA16\_ID**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__sbddata16</code> static and global variables in the <code>SBDATA16_I</code> segment. These values are copied from <code>SBDATA16_ID</code> to <code>SBDATA16_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 command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Memory placement    | This segment can be placed anywhere in ROM memory.                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |

**SBDATA16\_N**

|                     |                                                                                                                                |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __sbddata16</code> variables.                                                          |
| Segment memory type | DATA                                                                                                                           |
| Memory placement    | This segment must be placed within the first 64 Kbytes of memory after the static base reference point ( <code>SBREF</code> ). |
| Access type         | Read/write                                                                                                                     |



## SBDATA16\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                        |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__sbddata16</code> static and global variables.</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 command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                   |
| Memory placement    | This segment must be placed within the first 64 Kbytes of memory after the static base reference point ( <code>SBREF</code> ).                                                                                                                                                                                                                         |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                             |

## SBDATA24\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__sbddata24</code> static and global initialized variables initialized by copying from the segment <code>SBDATA24_ID</code> at application startup. Also holds <code>__sbddata24</code> constant data.</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 command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Memory placement    | This segment must be placed within the first 16 Mbytes of memory after the static base reference point ( <code>SBREF</code> ).                                                                                                                                                                                                                                                                                                                                                               |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

## SBDATA24\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__sbddata24</code> static and global variables in the <code>SBDATA24_I</code> segment. These values are copied from <code>SBDATA24_ID</code> to <code>SBDATA24_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 command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

|                  |                                                    |
|------------------|----------------------------------------------------|
| Memory placement | This segment can be placed anywhere in ROM memory. |
| Access type      | Read-only                                          |

**SBDATA24\_N**

|                     |                                                                                                                 |
|---------------------|-----------------------------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __sbddata24</code> variables.                                           |
| Segment memory type | DATA                                                                                                            |
| Memory placement    | This segment must be placed within the first 16 Mbytes of memory after the static base reference point (SBREF). |
| Access type         | Read/write                                                                                                      |

**SBDATA24\_Z**

|                     |                                                                                                                                                                                                                                                                                                                                                        |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__sbddata24</code> static and global variables.</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 command file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                   |
| Memory placement    | This segment must be placed within the first 16 Mbytes of memory after the static base reference point (SBREF).                                                                                                                                                                                                                                        |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                             |

# Implementation-defined behavior

This chapter describes how the compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1*, and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

---

## Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

### Translation

#### Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

*filename*, *linenumber* *level*[*tag*]: *message*

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

## Environment

### Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 55.

### Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

## Identifiers

### Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

### Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

### Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

## Characters

### Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 60.

### **Bits per character in execution character set (5.2.4.2.1)**

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

### **Mapping of characters (6.1.3.4)**

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

### **Unrepresented character constants (6.1.3.4)**

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

### **Character constant with more than one character (6.1.3.4)**

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

### **Converting multibyte characters (6.1.3.4)**

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 60.

### **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 154, 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 156, 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 158, for information about *size\_t*.

**Conversion from/to pointers (6.3.4)**

See *Casting*, page 158, for information about casting of data pointers and function pointers.

**ptrdiff\_t (6.3.6, 7.1.1)**

See *ptrdiff\_t*, page 158, 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 154, 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:

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
```

```
memory
module_name
no_pch
once
__printf_args
public_equ
__scanf_args
section
STDC
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 63.

**Terminating newline character (7.9.2)**

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

**Blank lines (7.9.2)**

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

**Null characters appended to data written to binary streams (7.9.2)**

No null characters are appended to data written to binary streams.

**Files (7.9.3)**

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 59.

**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 59.

**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 59.

### **%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: error message*

### **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 62.

**system() (7.10.4.5)**

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 62.

**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 63.

**clock() (7.12.2.1)**

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 63.



# A

abort  
     implementation-defined behavior (DLIB) . . . . . 248  
     system termination . . . . . 54  
 absolute location  
     data, placing at (@) . . . . . 104  
     language support for . . . . . 165  
     #pragma location . . . . . 192  
 addressing. *See* memory types, data models,  
 and code models  
 algorithm (STL header file) . . . . . 221  
 alignment . . . . . 153  
     forcing stricter (#pragma data\_alignment) . . . . . 187  
     in structures (#pragma pack) . . . . . 194  
     in structures, causing problems . . . . . 101  
     of an object (\_\_ALIGNOF\_\_) . . . . . 165  
     of data types . . . . . 153  
     restrictions for inline assembler . . . . . 74  
 alignment (pragma directive) . . . . . 245  
 \_\_ALIGNOF\_\_ (operator) . . . . . 165  
 --align\_func (compiler option) . . . . . 130  
 anonymous structures . . . . . 101  
 anonymous symbols, creating . . . . . 167  
 application  
     building, overview of . . . . . 4  
     startup and termination . . . . . 52  
 architecture  
     of R32C . . . . . 11  
 ARGFRAME (assembler directive) . . . . . 84  
 arrays  
     designated initializers in . . . . . 168  
     global, accessing . . . . . 86  
     implementation-defined behavior . . . . . 243  
     incomplete at end of structs . . . . . 167  
     non-lvalue . . . . . 170  
     of incomplete types . . . . . 169  
     single-value initialization . . . . . 171  
 asm, \_\_asm (language extension) . . . . . 167

assembler code  
     calling from C . . . . . 74  
     calling from C++ . . . . . 76  
     inserting inline . . . . . 73  
 assembler directives  
     using in inline assembler code . . . . . 74  
 assembler instructions  
     inserting inline . . . . . 73  
     used for calling functions . . . . . 85  
 assembler labels, making public (--public\_equ) . . . . . 150  
 assembler language interface . . . . . 71  
     calling convention. *See* assembler code  
 assembler list file, generating . . . . . 140  
 assembler output file . . . . . 76  
 assembler, inline . . . . . 166  
 asserts . . . . . 64  
     implementation-defined behavior of, (DLIB) . . . . . 246  
     including in application . . . . . 214  
 assert.h (DLIB header file) . . . . . 219  
 atoll, C99 extension . . . . . 224  
 atomic operations . . . . . 24  
     \_\_monitor . . . . . 180  
 attributes  
     object . . . . . 176  
     type . . . . . 173  
 auto variables . . . . . 17–18  
     at function entrance . . . . . 79  
     programming hints for efficient code . . . . . 111  
     using in inline assembler code . . . . . 74

# B

Barr, Michael . . . . . xxv  
 baseaddr (pragma directive) . . . . . 245  
 \_\_BASE\_FILE\_\_ (predefined symbol) . . . . . 212  
 basic type names, using in preprocessor expressions  
 (--migration\_preprocessor\_extensions) . . . . . 142  
 binary streams (DLIB) . . . . . 247  
 bit negation . . . . . 113

|                                                          |          |
|----------------------------------------------------------|----------|
| bitfields                                                |          |
| data representation of . . . . .                         | 155      |
| hints . . . . .                                          | 99       |
| implementation-defined behavior of . . . . .             | 243      |
| non-standard types in . . . . .                          | 165      |
| specifying order of members (#pragma bitfields). . . . . | 186      |
| bitfields (pragma directive). . . . .                    | 186      |
| bold style, in this guide . . . . .                      | xxvii    |
| bool (data type). . . . .                                | 154      |
| adding support for in DLIB . . . . .                     | 219, 222 |
| making available in C code . . . . .                     | 223      |
| __break (intrinsic function). . . . .                    | 201      |
| BRK (assembler instruction). . . . .                     | 201      |
| bubble sort function, defined in stdlib.h . . . . .      | 224      |
| building_runtime (pragma directive). . . . .             | 245      |
| __BUILD_NUMBER__ (predefined symbol) . . . . .           | 212      |
| byte order, identifying (__LITTLE_ENDIAN__) . . . . .    | 213      |

## C

|                                                         |     |
|---------------------------------------------------------|-----|
| C and C++ linkage . . . . .                             | 78  |
| C/C++ calling convention. <i>See</i> calling convention |     |
| C header files . . . . .                                | 219 |
| call frame information . . . . .                        | 88  |
| in assembler list file . . . . .                        | 76  |
| in assembler list file (-IA) . . . . .                  | 140 |
| call stack. . . . .                                     | 88  |
| callee-save registers, stored on stack. . . . .         | 18  |
| calling convention                                      |     |
| C++, requiring C linkage . . . . .                      | 76  |
| in compiler. . . . .                                    | 77  |
| <i>See also</i> assembler code                          |     |
| calloc (library function) . . . . .                     | 19  |
| <i>See also</i> heap                                    |     |
| implementation-defined behavior of (DLIB) . . . . .     | 248 |
| can_instantiate (pragma directive) . . . . .            | 245 |
| cassert (DLIB header file). . . . .                     | 221 |
| cast operators                                          |     |
| in Extended EC++ . . . . .                              | 92  |

|                                                                                |     |
|--------------------------------------------------------------------------------|-----|
| missing from Embedded C++ . . . . .                                            | 92  |
| casting, pointers and integers . . . . .                                       | 158 |
| cctype (DLIB header file) . . . . .                                            | 221 |
| cerrno (DLIB header file) . . . . .                                            | 221 |
| cexit (system termination code) . . . . .                                      | 52  |
| placement in segment. . . . .                                                  | 37  |
| CFI (assembler directive) . . . . .                                            | 88  |
| cfi.m53 (CFI header example file) . . . . .                                    | 89  |
| cfloat (DLIB header file). . . . .                                             | 221 |
| char (data type). . . . .                                                      | 154 |
| changing default representation (--char_is_signed) . . . . .                   | 131 |
| signed and unsigned. . . . .                                                   | 155 |
| characters, implementation-defined behavior of . . . . .                       | 240 |
| character-based I/O. . . . .                                                   | 56  |
| overriding in runtime library . . . . .                                        | 49  |
| --char_is_signed (compiler option). . . . .                                    | 131 |
| CHECKSUM (segment) . . . . .                                                   | 227 |
| cinttypes (DLIB header file) . . . . .                                         | 221 |
| classes. . . . .                                                               | 93  |
| climits (DLIB header file). . . . .                                            | 222 |
| clock (DLIB header file) . . . . .                                             | 222 |
| clock (DLIB library function),<br>implementation-defined behavior of . . . . . | 249 |
| clock.c . . . . .                                                              | 63  |
| __close (library function) . . . . .                                           | 59  |
| cmath (DLIB header file) . . . . .                                             | 222 |
| code                                                                           |     |
| execution of . . . . .                                                         | 6   |
| interruption of execution . . . . .                                            | 23  |
| verifying linked result . . . . .                                              | 38  |
| code models . . . . .                                                          | 21  |
| calling functions in. . . . .                                                  | 85  |
| configuration . . . . .                                                        | 6   |
| controlling default placement of constants. . . . .                            | 21  |
| identifying (__CODE_MODEL__) . . . . .                                         | 212 |
| specifying on command line (--code_model). . . . .                             | 131 |
| code motion (compiler transformation). . . . .                                 | 110 |
| disabling (--no_code_motion) . . . . .                                         | 143 |
| code pointers. <i>See</i> function pointers                                    |     |
| code segments, used for placement. . . . .                                     | 37  |



- codeseg (pragma directive) . . . . . 245
- `__CODE_MODEL__` (predefined symbol). . . . . 212
- `--code_model` (compiler option) . . . . . 131
- `__code24` (extended keyword) . . . . . 177
- CODE24 (segment). . . . . 227
- `__code32` (extended keyword) . . . . . 178
- `__code32` (function pointer) . . . . . 157
- CODE32 (segment). . . . . 227
- command line options
  - See also* compiler options
  - part of compiler invocation syntax . . . . . 119
  - passing. . . . . 119
  - typographic convention . . . . . xxvi
- command prompt icon, in this guide. . . . . xxvii
- comments
  - after preprocessor directives. . . . . 171
  - C++ style, using in C code. . . . . 166
- common block (call frame information) . . . . . 89
- common subexpr elimination (compiler transformation) . 109
- disabling (`--no_cse`) . . . . . 143
- compilation date
  - exact time of (`__TIME__`) . . . . . 214
  - identifying (`__DATE__`) . . . . . 212
- compiler
  - environment variables . . . . . 120
  - invocation syntax . . . . . 119
  - output from . . . . . 121
- compiler listing, generating (`-l`). . . . . 140
- compiler object file
  - including debug information in (`--debug, -r`) . . . . . 132
- compiler optimization levels . . . . . 107
- compiler options . . . . . 125
  - passing to compiler . . . . . 119
  - reading from file (`-f`) . . . . . 139
  - specifying parameters . . . . . 127
  - summary . . . . . 128
  - syntax of . . . . . 125
  - `-l` . . . . . 75
  - `--warnings_affect_exit_code` . . . . . 122
- compiler platform, identifying . . . . . 213
- compiler subversion number . . . . . 214
- compiler transformations . . . . . 106
- compiler version number . . . . . 214
- compiling
  - from the command line . . . . . 4
  - syntax. . . . . 119
- complex numbers, supported in Embedded C++. . . . . 92
- complex (library header file). . . . . 220
- compound literals . . . . . 167
- computer style, typographic convention . . . . . xxvi
- configuration
  - basic project settings . . . . . 5
  - `__low_level_init` . . . . . 55
- configuration symbols, in library configuration files. . . . . 51
- consistency, module . . . . . 67
- const
  - declaring objects . . . . . 162
  - non-top level . . . . . 170
- constants
  - default placement in memory. . . . . 21
  - overriding default placement of . . . . . 14
  - placing in named segment . . . . . 187
- constseg (pragma directive) . . . . . 187
- const\_cast (cast operator) . . . . . 92
- contents, of this guide. . . . . xxiv
- conventions, used in this guide . . . . . xxvi
- copyright notice . . . . . ii
- `__core` (runtime model attribute). . . . . 68
- `__cplusplus` (predefined symbol) . . . . . 212
- cross call (compiler transformation) . . . . . 111
- csetjmp (DLIB header file) . . . . . 222
- csignal (DLIB header file) . . . . . 222
- cspy\_support (pragma directive). . . . . 245
- CSTACK (segment) . . . . . 228
  - example . . . . . 34
  - See also* stack
- CSTART (segment). . . . . 37, 228

|                                                        |      |
|--------------------------------------------------------|------|
| cstartup (system startup code) . . . . .               | 37   |
| customizing . . . . .                                  | 55   |
| overriding in runtime library . . . . .                | 49   |
| cstartup.s53 . . . . .                                 | 52   |
| cstdarg (DLIB header file) . . . . .                   | 222  |
| cstdbool (DLIB header file) . . . . .                  | 222  |
| cstddef (DLIB header file) . . . . .                   | 222  |
| cstdio (DLIB header file) . . . . .                    | 222  |
| cstdlib (DLIB header file) . . . . .                   | 222  |
| cstring (DLIB header file) . . . . .                   | 222  |
| ctime (DLIB header file) . . . . .                     | 222  |
| cctype.h (library header file) . . . . .               | 219  |
| added C functionality . . . . .                        | 223  |
| cwctype.h (library header file) . . . . .              | 222  |
| C++                                                    |      |
| <i>See also</i> Embedded C++ and Extended Embedded C++ |      |
| absolute location . . . . .                            | 105  |
| calling convention . . . . .                           | 76   |
| dynamic initialization in . . . . .                    | 38   |
| features excluded from EC++ . . . . .                  | 91   |
| header files . . . . .                                 | 220  |
| language extensions . . . . .                          | 96   |
| special function types . . . . .                       | 26   |
| static member variables . . . . .                      | 105  |
| support for . . . . .                                  | 3    |
| C++ names, in assembler code . . . . .                 | 77   |
| C++ objects, placing in memory type . . . . .          | 17   |
| C++ terminology . . . . .                              | xxvi |
| C++-style comments . . . . .                           | 166  |
| C-SPY                                                  |      |
| low-level interface . . . . .                          | 65   |
| STL container support . . . . .                        | 95   |
| C_INCLUDE (environment variable) . . . . .             | 120  |
| C99 standard, added functionality from . . . . .       | 222  |

## D

|                                          |     |
|------------------------------------------|-----|
| -D (compiler option) . . . . .           | 131 |
| --data_model (compiler option) . . . . . | 132 |

|                                               |               |
|-----------------------------------------------|---------------|
| data                                          |               |
| alignment of . . . . .                        | 153           |
| different ways of storing . . . . .           | 11            |
| located, declaring extern . . . . .           | 105           |
| placing . . . . .                             | 103, 188, 225 |
| at absolute location . . . . .                | 104           |
| representation of . . . . .                   | 153           |
| storage . . . . .                             | 11            |
| verifying linked result . . . . .             | 38            |
| data block (call frame information) . . . . . | 89            |
| data memory attributes, using . . . . .       | 14            |
| data models . . . . .                         | 12            |
| configuration . . . . .                       | 6             |
| Far . . . . .                                 | 13            |
| Huge . . . . .                                | 13            |
| identifying (__DATA_MODEL__) . . . . .        | 212           |
| Near . . . . .                                | 13            |
| data pointers . . . . .                       | 158           |
| data segments . . . . .                       | 31            |
| data types . . . . .                          | 154           |
| avoiding signed . . . . .                     | 99            |
| floating point . . . . .                      | 156           |
| in C++ . . . . .                              | 162           |
| integers . . . . .                            | 154           |
| dataseg (pragma directive) . . . . .          | 188           |
| data_alignment (pragma directive) . . . . .   | 187           |
| __DATA_MODEL__ (predefined symbol) . . . . .  | 212           |
| __data16 (extended keyword) . . . . .         | 178           |
| data16 (memory type) . . . . .                | 14            |
| DATA16_AC (segment) . . . . .                 | 228           |
| DATA16_AN (segment) . . . . .                 | 228           |
| DATA16_C (segment) . . . . .                  | 229           |
| DATA16_I (segment) . . . . .                  | 229           |
| DATA16_ID (segment) . . . . .                 | 229           |
| DATA16_N (segment) . . . . .                  | 230           |
| DATA16_Z (segment) . . . . .                  | 230           |
| __data24 (extended keyword) . . . . .         | 179           |
| data24 (memory type) . . . . .                | 14            |
| DATA24_AC (segment) . . . . .                 | 230           |

- DATA24\_AN (segment) . . . . . 230
- DATA24\_C (segment) . . . . . 231
- DATA24\_I (segment) . . . . . 231
- DATA24\_ID (segment) . . . . . 231
- DATA24\_N (segment) . . . . . 232
- DATA24\_Z (segment) . . . . . 232
- \_\_data32 (extended keyword) . . . . . 179
- \_\_data32 (data pointer) . . . . . 158
- data32 (memory type) . . . . . 14
- DATA32\_AC (segment) . . . . . 232
- DATA32\_AN (segment) . . . . . 232
- DATA32\_C (segment) . . . . . 233
- DATA32\_I (segment) . . . . . 233
- DATA32\_ID (segment) . . . . . 233
- DATA32\_N (segment) . . . . . 234
- DATA32\_Z (segment) . . . . . 234
- \_\_DATE\_\_ (predefined symbol) . . . . . 212
- date (library function), configuring support for . . . . . 63
- DCR (register)
  - getting the value of (\_\_get\_DCR\_register) . . . . . 202
  - writing a value to (\_\_set\_DCR\_register) . . . . . 206
- DCT (register)
  - getting the value of (\_\_get\_DCT\_register) . . . . . 202
  - writing a value to (\_\_set\_DCT\_register) . . . . . 206
- DC32 (assembler directive) . . . . . 74
- DDA (register)
  - getting the value of (\_\_get\_DDA\_register) . . . . . 202
  - writing a value to (\_\_set\_DDA\_register) . . . . . 206
- DDR (register)
  - getting the value of (\_\_get\_DDR\_register) . . . . . 202
  - writing a value to (\_\_set\_DDR\_register) . . . . . 206
- debug (compiler option) . . . . . 132
- debug information, including in object file . . . . . 132, 150
- declarations
  - empty . . . . . 171
  - in for loops . . . . . 166
  - Kernighan & Ritchie . . . . . 112
  - of functions . . . . . 78
- declarations and statements, mixing . . . . . 166
- declarators, implementation-defined behavior . . . . . 244
- define\_type\_info (pragma directive) . . . . . 245
- delay code, inserting . . . . . 201
- \_\_delay\_cycles (intrinsic function) . . . . . 201
- delete (keyword) . . . . . 19
- dependencies (compiler option) . . . . . 133
- deque (STL header file) . . . . . 221
- destructors and interrupts, using . . . . . 95
- diagnostic messages . . . . . 122
  - classifying as compilation errors . . . . . 134
  - classifying as compilation remarks . . . . . 134
  - classifying as compiler warnings . . . . . 135
  - disabling compiler warnings . . . . . 146
  - disabling wrapping of in compiler . . . . . 146
  - enabling compiler remarks . . . . . 150
  - listing all used by compiler . . . . . 135
  - suppressing in compiler . . . . . 134
- diagnostics\_tables (compiler option) . . . . . 135
- diag\_default (pragma directive) . . . . . 188
- diag\_error (compiler option) . . . . . 134
- diag\_error (pragma directive) . . . . . 189
- diag\_remark (compiler option) . . . . . 134
- diag\_remark (pragma directive) . . . . . 189
- diag\_suppress (compiler option) . . . . . 134
- diag\_suppress (pragma directive) . . . . . 189
- diag\_warning (compiler option) . . . . . 135
- diag\_warning (pragma directive) . . . . . 190
- DIFUNCT (segment) . . . . . 38, 234
- directives
  - function for static overlay . . . . . 84
  - pragma . . . . . 9, 185
- directory, specifying as parameter . . . . . 126
- \_\_disable\_interrupt (intrinsic function) . . . . . 201
- discard\_unused\_publics (compiler option) . . . . . 136
- disclaimer . . . . . ii
- DLIB . . . . . 7, 218
  - See also* runtime library
  - configuring . . . . . 136
  - reference information. *See* the online help system . . . 217

|                                                          |      |
|----------------------------------------------------------|------|
| --dlib_config (compiler option) . . . . .                | 136  |
| Dlib_defaults.h (library configuration file) . . . . .   | 51   |
| dlr32c <code>libname</code> .h . . . . .                 | 51   |
| DMD (register)                                           |      |
| getting the value of (__get_DMD_register) . . . . .      | 203  |
| writing a value to (__set_DMD_register) . . . . .        | 206  |
| document conventions . . . . .                           | xxvi |
| documentation, library . . . . .                         | 217  |
| domain errors, implementation-defined behavior . . . . . | 246  |
| --double (compiler option) . . . . .                     | 137  |
| double (data type) . . . . .                             | 156  |
| avoiding . . . . .                                       | 99   |
| configuring size of floating-point type . . . . .        | 6    |
| __double_size (runtime model attribute) . . . . .        | 69   |
| double_t, C99 extension . . . . .                        | 223  |
| do_not_instantiate (pragma directive) . . . . .          | 245  |
| DSA (register)                                           |      |
| getting the value of (__get_DSA_register) . . . . .      | 203  |
| writing a value to (__set_DSA_register) . . . . .        | 207  |
| DSR (register)                                           |      |
| getting the value of (__get_DSR_register) . . . . .      | 203  |
| writing a value to (__set_DSR_register) . . . . .        | 207  |
| dynamic initialization . . . . .                         | 52   |
| in C++ . . . . .                                         | 38   |
| dynamic memory . . . . .                                 | 19   |

## E

|                                                   |     |
|---------------------------------------------------|-----|
| -e (compiler option) . . . . .                    | 137 |
| early_initialization (pragma directive) . . . . . | 245 |
| --ec++ (compiler option) . . . . .                | 137 |
| EC++ header files . . . . .                       | 220 |
| edition, of this guide . . . . .                  | ii  |
| --eec++ (compiler option) . . . . .               | 138 |
| Embedded C++ . . . . .                            | 91  |
| differences from C++ . . . . .                    | 91  |
| enabling . . . . .                                | 137 |
| function linkage . . . . .                        | 78  |
| language extensions . . . . .                     | 91  |

|                                                         |      |
|---------------------------------------------------------|------|
| overview . . . . .                                      | 91   |
| Embedded C++ Technical Committee . . . . .              | xxvi |
| embedded systems, IAR special support for . . . . .     | 8    |
| __embedded_cplusplus (predefined symbol) . . . . .      | 212  |
| __enable_interrupt (intrinsic function) . . . . .       | 201  |
| --enable_multibytes (compiler option) . . . . .         | 138  |
| entry label, program . . . . .                          | 52   |
| enumerations, implementation-defined behavior . . . . . | 243  |
| enums                                                   |      |
| data representation . . . . .                           | 154  |
| forward declarations of . . . . .                       | 169  |
| environment                                             |      |
| implementation-defined behavior . . . . .               | 240  |
| runtime . . . . .                                       | 41   |
| environment variables                                   |      |
| C_INCLUDE . . . . .                                     | 120  |
| QCCR32C . . . . .                                       | 120  |
| EQU (assembler directive) . . . . .                     | 150  |
| errno.h (library header file) . . . . .                 | 219  |
| error messages . . . . .                                | 123  |
| classifying for compiler . . . . .                      | 134  |
| error return codes . . . . .                            | 122  |
| --error_limit (compiler option) . . . . .               | 138  |
| exception handling, missing from Embedded C++ . . . . . | 91   |
| exception vectors . . . . .                             | 38   |
| exception (library header file) . . . . .               | 220  |
| __exchange_byte (intrinsic function) . . . . .          | 201  |
| __exchange_long (intrinsic function) . . . . .          | 202  |
| __exchange_word (intrinsic function) . . . . .          | 202  |
| _Exit (library function) . . . . .                      | 54   |
| exit (library function) . . . . .                       | 54   |
| implementation-defined behavior . . . . .               | 248  |
| _exit (library function) . . . . .                      | 54   |
| __exit (library function) . . . . .                     | 54   |
| export keyword, missing from Extended EC++ . . . . .    | 94   |
| extended command line file                              |      |
| for compiler . . . . .                                  | 139  |
| Extended Embedded C++ . . . . .                         | 92   |
| enabling . . . . .                                      | 138  |

standard template library (STL) . . . . . 221

extended keywords . . . . . 173

enabling (-e) . . . . . 137

overview . . . . . 8

summary . . . . . 177

syntax . . . . . 15

    object attributes . . . . . 176

    type attributes on data objects . . . . . 174

    type attributes on function pointers . . . . . 175

    type attributes on functions . . . . . 175

    \_\_code32 (function pointer) . . . . . 157

    \_\_data32 (data pointer) . . . . . 158

extern "C" linkage . . . . . 94

## F

-f (compiler option) . . . . . 139

Far (code model) . . . . . 22

\_\_fast\_interrupt (extended keyword) . . . . . 179

fatal error messages . . . . . 124

FB (register), considerations . . . . . 79

fgetpos (library function), implementation-defined behavior . . . . . 248

\_\_FILE\_\_ (predefined symbol) . . . . . 212

file dependencies, tracking . . . . . 133

file paths, specifying for #include files . . . . . 140

filenames, specifying as parameter . . . . . 126

FLG (register) . . . . . 201

float (data type) . . . . . 156

floating-point constants

    hexadecimal notation . . . . . 168

    hints . . . . . 100

floating-point expressions,

    using in preprocessor extensions . . . . . 142

floating-point format . . . . . 156

    casting to integer . . . . . 101

    hints . . . . . 100

    implementation-defined behavior . . . . . 242

    special cases . . . . . 157

    32-bits . . . . . 156

    64-bits . . . . . 157

floating-point implementation . . . . . 64

floating-point models . . . . . 64

    setting . . . . . 139

floating-point operations, hardware support for . . . . . 64

floating-point type, configuring size of double . . . . . 6

float.h (library header file) . . . . . 219

float\_t, C99 extension . . . . . 223

fmod (library function),

    implementation-defined behavior . . . . . 247

    for loops, declarations in . . . . . 166

formats

    floating-point values . . . . . 156

    standard IEEE (floating point) . . . . . 156

fpclassify, C99 extension . . . . . 223

FPU, IEEE-754 support . . . . . 64

fpu\_compliant.xcl (linker command file) . . . . . 65

FP\_INFINITE, C99 extension . . . . . 223

--fp\_model (compiler option) . . . . . 139

FP\_NAN, C99 extension . . . . . 223

FP\_NORMAL, C99 extension . . . . . 223

FP\_SUBNORMAL, C99 extension . . . . . 223

FP\_ZERO, C99 extension . . . . . 223

fragmentation, of heap memory . . . . . 19

frame pointer register, considerations . . . . . 79

free (library function). *See also* heap . . . . . 19

fstream (library header file) . . . . . 220

fstream.h (library header file) . . . . . 220

ftell (library function), implementation-defined behavior . 248

Full DLIB (library configuration) . . . . . 43

\_\_func\_\_ (predefined symbol) . . . . . 172, 213

FUNCALL (assembler directive) . . . . . 84

\_\_FUNCTION\_\_ (predefined symbol) . . . . . 172, 213

function calls

    calling convention . . . . . 77

    Far code model . . . . . 85

    Huge code model . . . . . 86

    stack image after . . . . . 81

function declarations, Kernighan & Ritchie . . . . . 112

|                                                                |                    |
|----------------------------------------------------------------|--------------------|
| function directives for static overlay . . . . .               | 84                 |
| function entry point, forcing alignment of . . . . .           | 82, 130            |
| function inlining (compiler transformation) . . . . .          | 109                |
| disabling (--no_inline) . . . . .                              | 144                |
| function pointers . . . . .                                    | 157                |
| function prototypes . . . . .                                  | 112                |
| enforcing . . . . .                                            | 151                |
| function return addresses . . . . .                            | 83                 |
| function type information, omitting in object output . . . . . | 148                |
| FUNCTION (assembler directive) . . . . .                       | 84                 |
| function (pragma directive) . . . . .                          | 245                |
| functional (STL header file) . . . . .                         | 221                |
| functions . . . . .                                            | 21                 |
| calling in different code models . . . . .                     | 85                 |
| C++ and special function types . . . . .                       | 26                 |
| declared without attribute, placement . . . . .                | 37                 |
| declaring . . . . .                                            | 78, 112            |
| inlining . . . . .                                             | 109, 111, 166, 191 |
| interrupt . . . . .                                            | 23–24              |
| intrinsic . . . . .                                            | 71, 112            |
| monitor . . . . .                                              | 24                 |
| omitting type info . . . . .                                   | 148                |
| parameters . . . . .                                           | 79                 |
| placing in memory . . . . .                                    | 103, 105           |
| recursive . . . . .                                            |                    |
| avoiding . . . . .                                             | 112                |
| storing data on stack . . . . .                                | 18–19              |
| reentrancy (DLIB) . . . . .                                    | 218                |
| related extensions . . . . .                                   | 21                 |
| return values from . . . . .                                   | 82                 |
| special function types . . . . .                               | 22                 |
| verifying linked result . . . . .                              | 38                 |

## G

|                                                               |     |
|---------------------------------------------------------------|-----|
| getenv (library function), configuring support for . . . . .  | 62  |
| getzone (library function), configuring support for . . . . . | 63  |
| getzone.c . . . . .                                           | 63  |
| __get_DCR_register (intrinsic function) . . . . .             | 202 |

|                                                      |       |
|------------------------------------------------------|-------|
| __get_DCT_register (intrinsic function) . . . . .    | 202   |
| __get_DDA_register (intrinsic function) . . . . .    | 202   |
| __get_DDR_register (intrinsic function) . . . . .    | 202   |
| __get_DMD_register (intrinsic function) . . . . .    | 203   |
| __get_DSA_register (intrinsic function) . . . . .    | 203   |
| __get_DSR_register (intrinsic function) . . . . .    | 203   |
| __get_interrupt_level (intrinsic function) . . . . . | 203   |
| __get_interrupt_state (intrinsic function) . . . . . | 203   |
| __get_interrupt_table (intrinsic function) . . . . . | 204   |
| __get_VCT_register (intrinsic function) . . . . .    | 204   |
| global arrays, accessing . . . . .                   | 86    |
| global variables . . . . .                           |       |
| accessing . . . . .                                  | 86    |
| initialization . . . . .                             | 33    |
| guidelines, reading . . . . .                        | xxiii |

## H

|                                              |          |
|----------------------------------------------|----------|
| Harbison, Samuel P. . . . .                  | xxvi     |
| hardware support in compiler . . . . .       | 42       |
| hash_map (STL header file) . . . . .         | 221      |
| hash_set (STL header file) . . . . .         | 221      |
| hdrstop (pragma directive) . . . . .         | 245      |
| header files . . . . .                       |          |
| C . . . . .                                  | 219      |
| C++ . . . . .                                | 220      |
| EC++ . . . . .                               | 220      |
| library . . . . .                            | 217      |
| special function registers . . . . .         | 114      |
| STL . . . . .                                | 221      |
| Dlib_defaults.h . . . . .                    | 51       |
| dlr32clibname.h . . . . .                    | 51       |
| intrinsics.h . . . . .                       | 199      |
| stdbool.h . . . . .                          | 154, 219 |
| stddef.h . . . . .                           | 155      |
| --header_context (compiler option) . . . . . | 140      |
| heap . . . . .                               |          |
| dynamic memory . . . . .                     | 19       |
| segment for . . . . .                        | 35       |

- storing data . . . . . 12
- heap segment
  - HEAP (segment) . . . . . 234
  - placing . . . . . 36
- heap size
  - and standard I/O . . . . . 36
  - changing default . . . . . 36
- HEAP (segment) . . . . . 36
- hints, optimization . . . . . 111
- Huge (code model) . . . . . 22
  - function calls . . . . . 86
- HUGE\_VALF, C99 extension . . . . . 223
- HUGE\_VALL, C99 extension . . . . . 223

## I

- I (compiler option) . . . . . 140
- IAR Command Line Build Utility . . . . . 51
- IAR Systems Technical Support . . . . . 124
- iarbuild.exe (utility) . . . . . 51
- \_\_IAR\_SYSTEMS\_ICC\_\_ (predefined symbol) . . . . . 213
- \_\_ICCR32C\_\_ (predefined symbol) . . . . . 213
- icons, in this guide . . . . . xxvii
- identifiers, implementation-defined behavior . . . . . 240
- IEEE format, floating-point values . . . . . 156
- \_\_illegal\_opcode (intrinsic function) . . . . . 204
- implementation-defined behavior . . . . . 239
- important\_typedef (pragma directive) . . . . . 245
- include files
  - including before source files . . . . . 149
  - specifying . . . . . 120
- include\_alias (pragma directive) . . . . . 190
- infinity . . . . . 157
- INFINITY, C99 extension . . . . . 223
- inheritance, in Embedded C++ . . . . . 91
- initialization
  - dynamic . . . . . 52
  - single-value . . . . . 171
- initialized data segments . . . . . 33

- initializers, static . . . . . 170
- inline assembler . . . . . 73, 166
  - avoiding . . . . . 112
  - See also* assembler language interface
- inline functions . . . . . 166
  - in compiler . . . . . 109
- inline (pragma directive) . . . . . 191
- instantiate (pragma directive) . . . . . 245
- INT (assembler instruction) . . . . . 208
- INTB (register)
  - getting the value of (\_\_get\_interrupt\_table) . . . . . 204
  - writing a value to (\_\_set\_interrupt\_table) . . . . . 207
- integer characteristics, adding . . . . . 223
- integers . . . . . 154
  - casting . . . . . 158
  - implementation-defined behavior . . . . . 242
  - intptr\_t . . . . . 158
  - ptrdiff\_t . . . . . 158
  - size\_t . . . . . 158
  - uintptr\_t . . . . . 158
- integral promotion . . . . . 113
- internal error . . . . . 124
- \_\_interrupt (extended keyword) . . . . . 23, 180
  - using in pragma directives . . . . . 198
- interrupt functions . . . . . 23
  - placement in memory . . . . . 38
- interrupt state, restoring . . . . . 207
- interrupt vector
  - specifying with pragma directive . . . . . 198
- interrupt vector table . . . . . 23
  - in linker command file . . . . . 38
- INTVEC segment . . . . . 235
- NMIVEC segment . . . . . 235
- interrupts
  - disabling . . . . . 180
    - during function execution . . . . . 24
  - processor state . . . . . 18
  - using with EC++ destructors . . . . . 95
- \_\_interrupt\_on\_overflow (intrinsic function) . . . . . 204

|                                             |            |
|---------------------------------------------|------------|
| INTO (assembler instruction) . . . . .      | 204        |
| intptr_t (integer type) . . . . .           | 158        |
| __intrinsic (extended keyword). . . . .     | 180        |
| intrinsic functions . . . . .               | 112        |
| overview . . . . .                          | 71         |
| summary . . . . .                           | 199        |
| __get_interrupt_state . . . . .             | 203        |
| intrinsics.h (header file) . . . . .        | 199        |
| inttypes.h (library header file). . . . .   | 219        |
| inttypes.h, added C functionality . . . . . | 223        |
| INTVEC (segment). . . . .                   | 38, 235    |
| invocation syntax . . . . .                 | 119        |
| iomanip (library header file) . . . . .     | 220        |
| iomanip.h (library header file) . . . . .   | 220        |
| ios (library header file) . . . . .         | 220        |
| iosfwd (library header file) . . . . .      | 220        |
| iostream (library header file). . . . .     | 220        |
| iostream.h (library header file) . . . . .  | 220        |
| isblank, C99 extension . . . . .            | 223        |
| isfinite, C99 extension . . . . .           | 223        |
| isgreater, C99 extension . . . . .          | 223        |
| isinf, C99 extension . . . . .              | 223        |
| islessequal, C99 extension . . . . .        | 223        |
| islessgreater, C99 extension . . . . .      | 223        |
| isless, C99 extension. . . . .              | 223        |
| isnan, C99 extension . . . . .              | 223        |
| isnormal, C99 extension . . . . .           | 223        |
| ISO/ANSI C . . . . .                        |            |
| compiler extensions . . . . .               | 163        |
| C++ features excluded from EC++ . . . . .   | 91         |
| specifying strict usage . . . . .           | 151        |
| iso646.h (library header file). . . . .     | 219        |
| ISP (stack pointer) . . . . .               | 235        |
| ISTACK (segment) . . . . .                  | 235        |
| <i>See also</i> stack . . . . .             |            |
| istream (library header file). . . . .      | 220        |
| isunordered, C99 extension. . . . .         | 223        |
| iswblank, C99 extension . . . . .           | 224        |
| italic style, in this guide . . . . .       | xxvi–xxvii |

|                                                     |     |
|-----------------------------------------------------|-----|
| iterator (STL header file) . . . . .                | 221 |
| I/O debugging, support for . . . . .                | 65  |
| I/O module, overriding in runtime library . . . . . | 49  |

## K

|                                                     |      |
|-----------------------------------------------------|------|
| keep_definition (pragma directive) . . . . .        | 245  |
| Kernighan & Ritchie function declarations . . . . . | 112  |
| disallowing. . . . .                                | 151  |
| Kernighan, Brian W. . . . .                         | xxvi |
| keywords . . . . .                                  |      |
| extended. . . . .                                   | 8    |

## L

|                                         |         |
|-----------------------------------------|---------|
| -l (compiler option). . . . .           | 75, 140 |
| labels. . . . .                         | 171     |
| assembler, making public. . . . .       | 150     |
| __program_start. . . . .                | 52      |
| Labrosse, Jean J. . . . .               | xxvi    |
| Lajoie, Josée . . . . .                 | xxvi    |
| language extensions . . . . .           |         |
| descriptions . . . . .                  | 163     |
| Embedded C++ . . . . .                  | 91      |
| enabling (compiler option). . . . .     | 137     |
| enabling (pragma directive) . . . . .   | 191     |
| language overview . . . . .             | 3       |
| language (pragma directive) . . . . .   | 191     |
| LDCTX (assembler instruction) . . . . . | 204     |
| libraries . . . . .                     |         |
| building . . . . .                      | 43      |
| definition of . . . . .                 | 4       |
| runtime. . . . .                        | 44      |
| standard template library . . . . .     | 221     |
| library configuration files . . . . .   |         |
| Dlib_defaults.h. . . . .                | 51      |
| dlr32clibname.h. . . . .                | 51      |
| modifying . . . . .                     | 51      |
| runtime library . . . . .               | 43      |



- specifying ..... 136
- library documentation ..... 217
- library features, missing from Embedded C++ ..... 92
- library functions ..... 217
  - reference information ..... xxv
  - summary, DLIB ..... 219
- library header files ..... 217
- library modules
  - creating ..... 141
  - overriding ..... 49
- library object files ..... 217
- library options, setting ..... 8
- library project template ..... 7, 51
- library\_module (compiler option) ..... 141
- lightbulb icon, in this guide ..... xxvii
- limits.h (library header file) ..... 219
- \_\_LINE\_\_ (predefined symbol) ..... 213
- linkage, C and C++ ..... 78
- linker command files ..... 28
  - customizing ..... 28
  - fpu\_compliant.xcl ..... 65
  - using the -P command in ..... 30
  - using the -Z command in ..... 30
- linker map file ..... 39
- linker segment. *See* segment
- linking
  - from the command line ..... 5
  - required input ..... 4
- Lippman, Stanley B. .... xxvi
- list (STL header file) ..... 221
- listing, generating ..... 140
- literals, compound ..... 167
- literature, recommended ..... xxv
- \_\_LITTLE\_ENDIAN\_\_ (predefined symbol) ..... 213
- llabs, C99 extension ..... 224
- lldiv, C99 extension ..... 224
- \_\_load\_context (intrinsic function) ..... 204
- local variables, *See* auto variables

- locale support ..... 60
  - adding ..... 61
  - changing at runtime ..... 61
  - removing ..... 61
- locale.h (library header file) ..... 219
- located data segments ..... 37
- located data, declaring extern ..... 105
- location (pragma directive) ..... 104, 192
- LOCFRAME (assembler directive) ..... 84
- long double (data type) ..... 156
- long float (data type), synonym for double ..... 170
- long long (data type), avoiding ..... 99
- loop overhead, reducing ..... 146
- loop unrolling (compiler transformation) ..... 109
  - disabling ..... 146
- loop-invariant expressions ..... 110
- low-level processor operations ..... 163, 199
  - accessing ..... 71
- \_\_low\_level\_init ..... 53
  - customizing ..... 55
- \_\_low\_level\_init (intrinsic function) ..... 205
- low\_level\_init.c ..... 52
- \_\_lseek (library function) ..... 59

## M

- macros
  - variadic ..... 215
- main (function), definition ..... 240
- malloc (library function)
  - See also* heap ..... 19
  - implementation-defined behavior ..... 248
- Mann, Bernhard ..... xxvi
- map (STL header file) ..... 221
- map, linker ..... 39
- math.h (library header file) ..... 219
- math.h, added C functionality ..... 223
- MATH\_ERREXCEPT, C99 extension ..... 223
- math\_errhandling, C99 extension ..... 223

|                                              |           |
|----------------------------------------------|-----------|
| MATH_ERRNO, C99 extension . . . . .          | 223       |
| memory . . . . .                             |           |
| accessing . . . . .                          | 6, 13, 86 |
| using data16 method . . . . .                | 87        |
| using data24 method . . . . .                | 87        |
| using data32 method . . . . .                | 87        |
| using sbdata16 method . . . . .              | 88        |
| using sbdata24 method . . . . .              | 88        |
| allocating in C++ . . . . .                  | 19        |
| dynamic . . . . .                            | 19        |
| heap . . . . .                               | 19        |
| non-initialized . . . . .                    | 115       |
| RAM, saving . . . . .                        | 112       |
| releasing in C++ . . . . .                   | 19        |
| stack . . . . .                              | 17        |
| saving . . . . .                             | 112       |
| used by global or static variables . . . . . | 11        |
| memory layout, R32C . . . . .                | 11        |
| memory management, type-safe . . . . .       | 91        |
| memory placement . . . . .                   |           |
| using pragma directive . . . . .             | 15        |
| using type definitions . . . . .             | 16, 175   |
| memory segment. <i>See</i> segment           |           |
| memory types . . . . .                       | 13        |
| C++ . . . . .                                | 17        |
| data16 . . . . .                             | 14        |
| data24 . . . . .                             | 14        |
| data32 . . . . .                             | 14        |
| placing variables in . . . . .               | 17        |
| sbdata16 . . . . .                           | 14        |
| sbdata24 . . . . .                           | 14        |
| specifying . . . . .                         | 14        |
| structures . . . . .                         | 16        |
| summary . . . . .                            | 15        |
| memory (pragma directive) . . . . .          | 246       |
| memory (STL header file) . . . . .           | 221       |
| message (pragma directive) . . . . .         | 192       |
| messages . . . . .                           |           |
| disabling . . . . .                          | 151       |

|                                                                 |          |
|-----------------------------------------------------------------|----------|
| forcing . . . . .                                               | 192      |
| --mfc (compiler option) . . . . .                               | 141      |
| --migration_preprocessor_extensions (compiler option) . . . . . | 142      |
| --misrac (compiler option) . . . . .                            | 129      |
| --misrac_verbose (compiler option) . . . . .                    | 129      |
| --misrac1998 (compiler option) . . . . .                        | 129      |
| --misrac2004 (compiler option) . . . . .                        | 129      |
| module consistency . . . . .                                    | 67       |
| rtmodel . . . . .                                               | 196      |
| module map, in linker map file . . . . .                        | 39       |
| module name, specifying . . . . .                               | 142      |
| module summary, in linker map file . . . . .                    | 39       |
| --module_name (compiler option) . . . . .                       | 142      |
| module_name (pragma directive) . . . . .                        | 246      |
| __monitor (extended keyword) . . . . .                          | 114, 180 |
| monitor functions . . . . .                                     | 24, 180  |
| multibyte character support . . . . .                           | 138      |
| multiple inheritance, missing from Embedded C++ . . . . .       | 91       |
| multi-file compilation . . . . .                                | 107      |
| mutable attribute, in Extended EC++ . . . . .                   | 92, 95   |

## N

|                                                |         |
|------------------------------------------------|---------|
| names block (call frame information) . . . . . | 89      |
| namespace support . . . . .                    |         |
| in Extended EC++ . . . . .                     | 92, 95  |
| missing from Embedded C++ . . . . .            | 92      |
| naming conventions . . . . .                   | xxvii   |
| NAN, C99 extension . . . . .                   | 223     |
| NDEBUG (preprocessor symbol) . . . . .         | 214     |
| new (keyword) . . . . .                        | 19      |
| new (library header file) . . . . .            | 220     |
| new.h (library header file) . . . . .          | 220     |
| NMIVEC (segment) . . . . .                     | 38, 235 |
| non-initialized variables, hints for . . . . . | 115     |
| non-scalar parameters, avoiding . . . . .      | 112     |
| NOP (assembler instruction) . . . . .          | 205     |
| __noreturn (extended keyword) . . . . .        | 181     |
| Normal DLIB (library configuration) . . . . .  | 43      |

Not a number (NaN) . . . . . 64–65, 157  
 --no\_code\_motion (compiler option) . . . . . 143  
 --no\_cse (compiler option) . . . . . 143  
 \_\_no\_init (extended keyword) . . . . . 115, 181  
 --no\_inline (compiler option) . . . . . 144  
 \_\_no\_operation (intrinsic function) . . . . . 205  
 --no\_path\_in\_file\_macros (compiler option) . . . . . 144  
 no\_pch (pragma directive) . . . . . 246  
 --no\_typedefs\_in\_diagnostics (compiler option) . . . . . 145  
 --no\_unroll (compiler option) . . . . . 146  
 --no\_warnings (compiler option) . . . . . 146  
 --no\_wrap\_diagnostics (compiler option) . . . . . 146  
 NULL (macro), implementation-defined behavior . . . . . 246  
 numeric (STL header file) . . . . . 221

## O

-O (compiler option) . . . . . 147  
 -o (compiler option) . . . . . 147–148  
 object attributes . . . . . 176  
 object filename  
     specifying in compiler . . . . . 147–148  
 object module name, specifying . . . . . 142  
 object\_attribute (pragma directive) . . . . . 115, 193  
 --omit\_types (compiler option) . . . . . 148  
 once (pragma directive) . . . . . 246  
 --only\_stdout (compiler option) . . . . . 148  
 \_\_open (library function) . . . . . 59  
 operators  
     *See also* @ (operator)  
 optimization  
     code motion, disabling . . . . . 143  
     common sub-expression elimination, disabling . . . . . 143  
     configuration . . . . . 7  
     disabling . . . . . 109  
     function inlining, disabling (--no\_inline) . . . . . 144  
     hints . . . . . 111  
     loop unrolling, disabling . . . . . 146  
     specifying (-O) . . . . . 147

summary . . . . . 107  
 techniques . . . . . 109  
 type-based alias analysis (compiler option) . . . . . 110  
     disabling . . . . . 145  
     using inline assembler code . . . . . 74  
     using pragma directive . . . . . 193  
 optimization levels . . . . . 107  
 optimize (pragma directive) . . . . . 193  
 option parameters . . . . . 125  
 options, compiler. *See* compiler options  
 Oram, Andy . . . . . xxv  
 ostream (library header file) . . . . . 220  
 output files, from XLINK . . . . . 5  
 output (preprocessor) . . . . . 149  
 overhead, reducing . . . . . 109

## P

pack (pragma directive) . . . . . 159, 194  
 \_\_packed (extended keyword) . . . . . 181  
 packed structure types . . . . . 159  
 parameters  
     function . . . . . 79  
     hidden . . . . . 80  
     non-scalar, avoiding . . . . . 112  
     register . . . . . 79–80  
     rules for specifying a file or directory . . . . . 126  
     specifying . . . . . 127  
     stack . . . . . 79, 81  
 parameters, typographic convention . . . . . xxvi  
 part number, of this guide . . . . . ii  
 permanent registers . . . . . 79  
 perror (library function),  
   implementation-defined behavior . . . . . 248  
 placement  
     code and data . . . . . 225  
     in named segments . . . . . 105  
 pointer types . . . . . 157  
     mixing . . . . . 170

|                                            |          |
|--------------------------------------------|----------|
| pointers                                   |          |
| casting                                    | 158      |
| data                                       | 158      |
| function                                   | 157      |
| implementation-defined behavior            | 243      |
| polymorphism, in Embedded C++              | 91       |
| porting, code containing pragma directives | 186      |
| __Pragma (predefined symbol)               | 215      |
| pragma directives                          | 9        |
| summary                                    | 185      |
| bitfields                                  | 155      |
| for absolute located data                  | 104      |
| list of all recognized                     | 245      |
| pack                                       | 159, 194 |
| type_attribute, using                      | 15       |
| predefined symbols                         |          |
| overview                                   | 9        |
| summary                                    | 212      |
| --predef_macro (compiler option)           | 149      |
| --preinclude (compiler option)             | 149      |
| --preprocess (compiler option)             | 149      |
| preprocessor                               |          |
| output                                     | 149      |
| overview                                   | 211      |
| preprocessor directives                    |          |
| implementation-defined behavior            | 244      |
| preprocessor extensions                    |          |
| compatibility                              | 142      |
| #warning message                           | 215      |
| __VA_ARGS__                                | 215      |
| preprocessor symbols                       | 212      |
| defining                                   | 131      |
| preserved registers                        | 79       |
| __PRETTY_FUNCTION__ (predefined symbol)    | 213      |
| primitives, for special functions          | 22       |
| print formatter, selecting                 | 48       |
| printf (library function)                  | 47       |
| choosing formatter                         | 47       |
| configuration symbols                      | 58       |

|                                             |          |
|---------------------------------------------|----------|
| implementation-defined behavior             | 248      |
| processor operations                        |          |
| accessing                                   | 71       |
| low-level                                   | 163, 199 |
| program entry label                         | 52       |
| programming hints                           | 111      |
| __program_start (label)                     | 52       |
| projects                                    |          |
| basic settings for                          | 5        |
| prototypes, enforcing                       | 151      |
| ptrdiff_t (integer type)                    | 158      |
| PUBLIC (assembler directive)                | 150      |
| publication date, of this guide             | ii       |
| --public_equ (compiler option)              | 150      |
| public_equ (pragma directive)               | 246      |
| putenv (library function), absent from DLIB | 62       |

## Q

|                                 |     |
|---------------------------------|-----|
| QCCR32C (environment variable)  | 120 |
| qualifiers                      |     |
| implementation-defined behavior | 244 |
| queue (STL header file)         | 221 |

## R

|                                                   |       |
|---------------------------------------------------|-------|
| -r (compiler option)                              | 150   |
| raise (library function), configuring support for | 63    |
| raise.c                                           | 63    |
| RAM                                               |       |
| non-zero initialized variables                    | 33    |
| saving memory                                     | 112   |
| range errors, in linker                           | 38    |
| __read (library function)                         | 59    |
| customizing                                       | 56    |
| read formatter, selecting                         | 49    |
| reading guidelines                                | xxiii |
| reading, recommended                              | xxv   |

- realloc (library function)
  - implementation-defined behavior. . . . . 248
  - See also* heap . . . . . 19
- recursive functions
  - avoiding . . . . . 112
  - storing data on stack . . . . . 18–19
- reentrancy (DLIB). . . . . 218
- reference information, typographic convention. . . . . xxvii
- register parameters . . . . . 79–80
- registered trademarks . . . . . ii
- registers
  - assigning to parameters . . . . . 80
  - callee-save, stored on stack . . . . . 18
  - for function returns . . . . . 83
  - implementation-defined behavior. . . . . 243
  - in assembler-level routines. . . . . 77
  - preserved . . . . . 79
  - scratch . . . . . 79
  - DCR, getting the value of (`__get_DCR_register`). . . . . 202
  - DCR, writing a value to (`__set_DCR_register`). . . . . 206
  - DCT, getting the value of (`__get_DCT_register`). . . . . 202
  - DCT, writing a value to (`__set_DCT_register`). . . . . 206
  - DDA, getting the value of (`__get_DDA_register`). . . . . 202
  - DDA, writing a value to (`__set_DDA_register`). . . . . 206
  - DDR, getting the value of (`__get_DDR_register`). . . . . 202
  - DDR, writing a value to (`__set_DDR_register`). . . . . 206
  - DMD, getting the value of (`__get_DMD_register`). . . . . 203
  - DMD, writing a value to (`__set_DMD_register`). . . . . 206
  - DSA, getting the value of (`__get_DSA_register`). . . . . 203
  - DSA, writing a value to (`__set_DSA_register`). . . . . 207
  - DSR, getting the value of (`__get_DSR_register`). . . . . 203
  - DSR, writing a value to (`__set_DSR_register`). . . . . 207
  - INTB, getting the value of (`__get_interrupt_table`). . . . . 204
  - INTB, writing a value to (`__set_interrupt_table`). . . . . 207
  - VCT, getting the value of (`__get_VCT_register`). . . . . 204
  - VCT, writing a value to (`__set_VCT_register`). . . . . 207
- reinterpret\_cast (cast operator) . . . . . 92
- remark (diagnostic message)
  - classifying for compiler . . . . . 134
  - enabling in compiler . . . . . 150
  - remarks (compiler option) . . . . . 150
- remarks (diagnostic message). . . . . 123
- remove (library function) . . . . . 59
  - implementation-defined behavior. . . . . 247
- rename (library function) . . . . . 59
  - implementation-defined behavior. . . . . 247
- `__ReportAssert` (library function). . . . . 64
- required (pragma directive). . . . . 195
- require\_prototypes (compiler option). . . . . 151
- return addresses . . . . . 83
- return values, from functions . . . . . 82
- Ritchie, Dennis M. . . . . xxvi
- RMPA.B (assembler instruction) . . . . . 205
- RMPA.L (assembler instruction). . . . . 205
- RMPA.W (assembler instruction) . . . . . 205
- `__RMPA_B` (intrinsic function) . . . . . 205
- `__RMPA_L` (intrinsic function) . . . . . 205
- `__RMPA_W` (intrinsic function). . . . . 205
- `__root` (extended keyword) . . . . . 182
- ROUND (assembler instruction). . . . . 206
- `__ROUND` (intrinsic function) . . . . . 101, 206
- routines, time-critical . . . . . 71, 163, 199
- RTMODEL (assembler directive) . . . . . 68
- rtmodel (pragma directive) . . . . . 196
- rtti support, missing from STL . . . . . 92
- `__rt_version` (runtime model attribute) . . . . . 69
- runtime environment. . . . . 41
  - setting options . . . . . 8
- runtime libraries
  - building customized. . . . . 43
  - choosing. . . . . 8, 45
  - configurations . . . . . 43
  - configuring. . . . . 42
  - customizing without rebuilding . . . . . 46
  - debug support. . . . . 44
  - DLIB . . . . . 44
  - overriding modules in . . . . . 49
  - introduction . . . . . 217

|                                                               |     |
|---------------------------------------------------------------|-----|
| naming convention . . . . .                                   | 45  |
| runtime model attributes . . . . .                            | 67  |
| __core . . . . .                                              | 68  |
| __double_size . . . . .                                       | 69  |
| __rt_version . . . . .                                        | 69  |
| runtime model definitions . . . . .                           | 196 |
| runtime type information, missing from Embedded C++ . . . . . | 92  |
| R32C . . . . .                                                |     |
| memory access . . . . .                                       | 6   |
| memory layout . . . . .                                       | 11  |
| supported devices . . . . .                                   | 4   |
| R32C/100, instruction set . . . . .                           | 86  |

## S

|                                           |     |
|-------------------------------------------|-----|
| SB (register), considerations . . . . .   | 79  |
| __sbddata16 (extended keyword) . . . . .  | 182 |
| sbdata16 (memory type) . . . . .          | 14  |
| SBDATA16_I (segment) . . . . .            | 236 |
| SBDATA16_ID (segment) . . . . .           | 236 |
| SBDATA16_N (segment) . . . . .            | 236 |
| SBDATA16_Z (segment) . . . . .            | 237 |
| __sbddata24 (extended keyword) . . . . .  | 183 |
| sbdata24 (memory type) . . . . .          | 14  |
| SBDATA24_I (segment) . . . . .            | 237 |
| SBDATA24_ID (segment) . . . . .           | 237 |
| SBDATA24_N (segment) . . . . .            | 238 |
| SBDATA24_Z (segment) . . . . .            | 238 |
| SB_START (linker symbol) . . . . .        | 31  |
| scanf (library function) . . . . .        |     |
| choosing formatter . . . . .              | 48  |
| configuration symbols . . . . .           | 58  |
| implementation-defined behavior . . . . . | 248 |
| scratch registers . . . . .               | 79  |
| section (pragma directive) . . . . .      | 246 |
| segment group name . . . . .              | 32  |
| segment map . . . . .                     |     |
| in linker map file . . . . .              | 39  |
| segment memory types, in XLINK . . . . .  | 28  |

|                                      |     |
|--------------------------------------|-----|
| segment names, declaring . . . . .   | 197 |
| segment (pragma directive) . . . . . | 197 |
| segments . . . . .                   | 225 |
| code . . . . .                       | 37  |
| data . . . . .                       | 31  |
| definition of . . . . .              | 27  |
| initialized data . . . . .           | 33  |
| introduction . . . . .               | 27  |
| located data . . . . .               | 37  |
| naming . . . . .                     | 32  |
| packing in memory . . . . .          | 30  |
| placing in sequence . . . . .        | 30  |
| static memory . . . . .              | 31  |
| summary . . . . .                    | 225 |
| too long for address range . . . . . | 38  |
| too long, in linker . . . . .        | 38  |
| CODE24 . . . . .                     | 227 |
| CODE32 . . . . .                     | 227 |
| CSTACK . . . . .                     | 228 |
| CSTART . . . . .                     | 228 |
| DATA16_AC . . . . .                  | 228 |
| DATA16_AN . . . . .                  | 228 |
| DATA16_C . . . . .                   | 229 |
| DATA16_I . . . . .                   | 229 |
| DATA16_ID . . . . .                  | 229 |
| DATA16_N . . . . .                   | 230 |
| DATA16_Z . . . . .                   | 230 |
| DATA24_AC . . . . .                  | 230 |
| DATA24_AN . . . . .                  | 230 |
| DATA24_C . . . . .                   | 231 |
| DATA24_I . . . . .                   | 231 |
| DATA24_ID . . . . .                  | 231 |
| DATA24_N . . . . .                   | 232 |
| DATA24_Z . . . . .                   | 232 |
| DATA32_AC . . . . .                  | 232 |
| DATA32_AN . . . . .                  | 232 |
| DATA32_C . . . . .                   | 233 |
| DATA32_I . . . . .                   | 233 |
| DATA32_ID . . . . .                  | 233 |

- DATA32\_N ..... 234
- DATA32\_Z ..... 234
- INTVEC ..... 38
- ISTACK ..... 235
- NMIVEC ..... 38, 235
- SBDATA16\_I ..... 236
- SBDATA16\_ID ..... 236
- SBDATA16\_N ..... 236
- SBDATA16\_Z ..... 237
- SBDATA24\_I ..... 237
- SBDATA24\_ID ..... 237
- SBDATA24\_N ..... 238
- SBDATA24\_Z ..... 238
- \_\_segment\_begin (extended operator) ..... 165
- \_\_segment\_end (extended operator) ..... 165
- semaphores
  - C example ..... 24
  - C++ example ..... 25
  - operations on ..... 180
- set (STL header file) ..... 221
- setjmp.h (library header file) ..... 219
- setlocale (library function) ..... 61
- settings, basic for project configuration ..... 5
- \_\_set\_DCR\_register (intrinsic function) ..... 206
- \_\_set\_DCT\_register (intrinsic function) ..... 206
- \_\_set\_DDA\_register (intrinsic function) ..... 206
- \_\_set\_DDR\_register (intrinsic function) ..... 206
- \_\_set\_DMD\_register (intrinsic function) ..... 206
- \_\_set\_DSA\_register (intrinsic function) ..... 207
- \_\_set\_DSR\_register (intrinsic function) ..... 207
- \_\_set\_interrupt\_level (intrinsic function) ..... 207
- \_\_set\_interrupt\_state (intrinsic function) ..... 207
- \_\_set\_interrupt\_table (intrinsic function) ..... 207
- \_\_set\_VCT\_register (intrinsic function) ..... 207
- severity level
  - of diagnostic messages ..... 123
  - specifying ..... 124
- SFR (special function registers) ..... 114
  - declaring extern ..... 105
- shared object ..... 122
- short (data type) ..... 154
- signal (library function)
  - configuring support for ..... 63
  - implementation-defined behavior ..... 247
- signal.c ..... 63
- signal.h (library header file) ..... 219
- signbit, C99 extension ..... 223
- signed char (data type) ..... 154–155
  - specifying ..... 131
- signed int (data type) ..... 154
- signed long long (data type) ..... 154
- signed long (data type) ..... 154
- signed short (data type) ..... 154
- signed values, avoiding ..... 99
- silent (compiler option) ..... 151
- silent operation
  - specifying in compiler ..... 151
- \_\_SIN (intrinsic function) ..... 208
- SIN (assembler instruction) ..... 208
- 64-bits (floating-point format) ..... 157
- sizeof, using in preprocessor extensions ..... 142
- size\_t (integer type) ..... 158
- skeleton code
  - creating for assembler language interface ..... 74
- skeleton.s53 (assembler source output) ..... 76
- slist (STL header file) ..... 221
- snprintf, C99 extension ..... 223
- \_\_software\_interrupt (intrinsic function) ..... 208
- source files, list all referred ..... 140
- \_\_SOUT (intrinsic function) ..... 208
- SOUT.B (assembler instruction) ..... 208
- special function registers (SFR) ..... 114
- special function types ..... 22
  - overview ..... 9
- sprintf (library function) ..... 47
  - choosing formatter ..... 47
- sscanf (library function)
  - choosing formatter ..... 48

|                                                       |             |
|-------------------------------------------------------|-------------|
| sstream (library header file) . . . . .               | 220         |
| stack . . . . .                                       | 17, 34      |
| advantages and problems using . . . . .               | 18          |
| changing default size of . . . . .                    | 34          |
| cleaning after function return . . . . .              | 83          |
| contents of . . . . .                                 | 18          |
| for interrupts . . . . .                              | 35          |
| internal data . . . . .                               | 228         |
| interrupt . . . . .                                   | 235         |
| layout . . . . .                                      | 81          |
| saving space . . . . .                                | 112         |
| size . . . . .                                        | 35          |
| stack parameters . . . . .                            | 79, 81      |
| stack pointer . . . . .                               | 18          |
| stack pointer register, considerations . . . . .      | 79          |
| stack segment . . . . .                               |             |
| placing in memory . . . . .                           | 35          |
| stack (STL header file) . . . . .                     | 221         |
| standard error . . . . .                              |             |
| redirecting in compiler . . . . .                     | 148         |
| standard input . . . . .                              | 56          |
| standard output . . . . .                             | 56          |
| specifying in compiler . . . . .                      | 148         |
| standard template library (STL) . . . . .             |             |
| in Extended EC++ . . . . .                            | 92, 95, 221 |
| missing from Embedded C++ . . . . .                   | 92          |
| startup code . . . . .                                |             |
| placement of . . . . .                                | 37          |
| <i>See also</i> CSTART . . . . .                      |             |
| startup, system . . . . .                             | 52          |
| statements, implementation-defined behavior . . . . . | 244         |
| static data, in linker command file . . . . .         | 34          |
| static memory segments . . . . .                      | 31          |
| static overlay . . . . .                              | 84          |
| static variables . . . . .                            | 11          |
| initialization . . . . .                              | 33          |
| taking the address of . . . . .                       | 111         |
| static_cast (cast operator) . . . . .                 | 92          |
| STCTX (assembler instruction) . . . . .               | 208         |

|                                                              |          |
|--------------------------------------------------------------|----------|
| std namespace, missing from EC++ . . . . .                   | 95       |
| and Extended EC++ . . . . .                                  | 95       |
| stdarg.h (library header file) . . . . .                     | 219      |
| stdbool.h (library header file) . . . . .                    | 154, 219 |
| added C functionality . . . . .                              | 223      |
| __STDC__ (predefined symbol) . . . . .                       | 213      |
| STDC (pragma directive) . . . . .                            | 246      |
| __STDC_VERSION__ (predefined symbol) . . . . .               | 213      |
| stddef.h (library header file) . . . . .                     | 155, 219 |
| stderr . . . . .                                             | 59, 148  |
| stdexcept (library header file) . . . . .                    | 220      |
| stdin . . . . .                                              | 59       |
| implementation-defined behavior . . . . .                    | 247      |
| stdint.h (library header file) . . . . .                     | 219, 222 |
| stdint.h, added C functionality . . . . .                    | 223      |
| stdio.h (library header file) . . . . .                      | 219      |
| stdio.h, additional C functionality . . . . .                | 223      |
| stdlib.h (library header file) . . . . .                     | 219      |
| stdlib.h, additional C functionality . . . . .               | 224      |
| stdout . . . . .                                             | 59, 148  |
| implementation-defined behavior . . . . .                    | 247      |
| Steele, Guy L. . . . .                                       | xxvi     |
| STL . . . . .                                                | 95       |
| __STOP (intrinsic function) . . . . .                        | 208      |
| STOP (assembler instruction) . . . . .                       | 208      |
| __store_context (intrinsic function) . . . . .               | 208      |
| streambuf (library header file) . . . . .                    | 220      |
| streams, supported in Embedded C++ . . . . .                 | 92       |
| strerror (library function) . . . . .                        |          |
| implementation-defined behavior . . . . .                    | 249      |
| --strict_ansi (compiler option) . . . . .                    | 151      |
| string (library header file) . . . . .                       | 220      |
| strings, supported in Embedded C++ . . . . .                 | 92       |
| string.h (library header file) . . . . .                     | 219      |
| Stroustrup, Bjarne . . . . .                                 | xxvi     |
| strstream (library header file) . . . . .                    | 220      |
| strtod (library function), configuring support for . . . . . | 63       |
| strtod, in stdlib.h . . . . .                                | 224      |
| strtof, C99 extension . . . . .                              | 224      |
| strtold, C99 extension . . . . .                             | 224      |



|                                                |             |
|------------------------------------------------|-------------|
| strtol, C99 extension                          | 224         |
| strtoull, C99 extension                        | 224         |
| structure types                                |             |
| alignment                                      | 159         |
| layout of                                      | 159         |
| packed                                         | 159         |
| structures                                     |             |
| accessing using a pointer                      | 86          |
| aligning                                       | 194         |
| anonymous                                      | 101, 165    |
| implementation-defined behavior                | 243         |
| incomplete arrays as last element              | 167         |
| packing and unpacking                          | 101         |
| placing in memory type                         | 16          |
| subnormal numbers                              | 156         |
| __SUBVERSION__ (predefined symbol)             | 214         |
| support, technical                             | 124         |
| symbol names, using in preprocessor extensions | 142         |
| symbols                                        |             |
| anonymous, creating                            | 167         |
| including in output                            | 195         |
| listing in linker map file                     | 39          |
| overview of predefined                         | 9           |
| preprocessor, defining                         | 131         |
| syntax                                         |             |
| command line options                           | 125         |
| extended keywords                              | 15, 174–176 |
| invoking compiler                              | 119         |
| system startup                                 | 52          |
| customizing                                    | 55          |
| system termination                             | 54          |
| C-SPY interface to                             | 55          |
| system (library function)                      |             |
| configuring support for                        | 62          |
| implementation-defined behavior                | 249         |
| system_include (pragma directive)              | 246         |

## T

|                                                      |              |
|------------------------------------------------------|--------------|
| __task (extended keyword)                            | 183          |
| technical support, IAR Systems                       | 124          |
| template support                                     |              |
| in Extended EC++                                     | 92, 94       |
| missing from Embedded C++                            | 91           |
| Terminal I/O window, making available                | 66           |
| terminal output, speeding up                         | 67           |
| termination, of system                               | 54           |
| terminology                                          | xxvi         |
| 32-bits (floating-point format)                      | 156          |
| this (pointer)                                       | 76           |
| referring to a class object                          | 94           |
| __TIME__ (predefined symbol)                         | 214          |
| time zone (library function)                         |              |
| implementation-defined behavior                      | 249          |
| time (library function), configuring support for     | 63           |
| time-critical routines                               | 71, 163, 199 |
| time.c                                               | 63           |
| time.h (library header file)                         | 219          |
| tips, programming                                    | 111          |
| tools icon, in this guide                            | xxvii        |
| trademarks                                           | ii           |
| transformations, compiler                            | 106          |
| translation, implementation-defined behavior         | 239          |
| trap vectors, specifying with pragma directive       | 198          |
| type attributes                                      | 173          |
| specifying                                           | 197          |
| type definitions, used for specifying memory storage | 16, 175      |
| type information, omitting                           | 148          |
| type qualifiers                                      |              |
| const and volatile                                   | 161          |
| implementation-defined behavior                      | 244          |
| typedefs                                             |              |
| excluding from diagnostics                           | 145          |
| repeated                                             | 170          |
| using in preprocessor extensions                     | 142          |

|                                                             |         |
|-------------------------------------------------------------|---------|
| type-based alias analysis (compiler transformation) . . . . | 110     |
| disabling . . . . .                                         | 145     |
| type-safe memory management . . . . .                       | 91      |
| type_attribute (pragma directive) . . . . .                 | 15, 197 |
| typographic conventions . . . . .                           | xxvi    |

## U

|                                                                      |          |
|----------------------------------------------------------------------|----------|
| uintptr_t (integer type) . . . . .                                   | 158      |
| UND (assembler instruction) . . . . .                                | 204      |
| underflow range errors,<br>implementation-defined behavior . . . . . | 246      |
| unions                                                               |          |
| anonymous. . . . .                                                   | 101, 165 |
| implementation-defined behavior. . . . .                             | 243      |
| unsigned char (data type) . . . . .                                  | 154–155  |
| changing to signed char . . . . .                                    | 131      |
| unsigned int (data type). . . . .                                    | 154      |
| unsigned long long (data type) . . . . .                             | 154      |
| unsigned long (data type) . . . . .                                  | 154      |
| unsigned short (data type). . . . .                                  | 154      |
| USER_RAM_BEGIN (linker symbol) . . . . .                             | 31       |
| USER_RAM_END (linker symbol) . . . . .                               | 31       |
| USER_ROM_BEGIN (linker symbol) . . . . .                             | 31       |
| USER_ROM_END (linker symbol) . . . . .                               | 31       |
| USP (stack pointer). . . . .                                         | 228      |
| utility (STL header file) . . . . .                                  | 221      |

## V

|                                                              |       |
|--------------------------------------------------------------|-------|
| variable type information, omitting in object output . . . . | 148   |
| variables                                                    |       |
| auto . . . . .                                               | 17–18 |
| defined inside a function . . . . .                          | 17    |
| global                                                       |       |
| accessing. . . . .                                           | 86    |
| placement in memory . . . . .                                | 11    |
| hints for choosing . . . . .                                 | 111   |
| local. <i>See</i> auto variables                             |       |

|                                                    |         |
|----------------------------------------------------|---------|
| non-initialized . . . . .                          | 115     |
| omitting type info . . . . .                       | 148     |
| placing at absolute addresses . . . . .            | 105     |
| placing in named segments . . . . .                | 105     |
| static                                             |         |
| placement in memory . . . . .                      | 11      |
| taking the address of . . . . .                    | 111     |
| static and global, initializing . . . . .          | 33      |
| VCT (register)                                     |         |
| getting the value of (__get_VCT_register). . . . . | 204     |
| writing a value to (__set_VCT_register) . . . . .  | 207     |
| vector (pragma directive) . . . . .                | 23, 198 |
| vector (STL header file) . . . . .                 | 221     |
| __VER__ (predefined symbol) . . . . .              | 214     |
| version                                            |         |
| compiler. . . . .                                  | 214     |
| IAR Embedded Workbench . . . . .                   | ii      |
| vfscanf, C99 extension . . . . .                   | 223     |
| vfwscanf, C99 extension. . . . .                   | 224     |
| void, pointers to . . . . .                        | 170     |
| volatile (keyword). . . . .                        | 113     |
| volatile, declaring objects . . . . .              | 161     |
| vscanf, C99 extension . . . . .                    | 223     |
| vsprintf, C99 extension . . . . .                  | 223     |
| vsscanf, C99 extension . . . . .                   | 223     |
| vswscanf, C99 extension. . . . .                   | 224     |
| vwsscanf, C99 extension . . . . .                  | 224     |

## W

|                                                    |       |
|----------------------------------------------------|-------|
| WAIT (assembler instruction). . . . .              | 209   |
| __wait_for_interrupt (intrinsic function). . . . . | 209   |
| #warning message (preprocessor extension). . . . . | 215   |
| warnings . . . . .                                 | 123   |
| classifying in compiler . . . . .                  | 135   |
| disabling in compiler . . . . .                    | 146   |
| exit code. . . . .                                 | 152   |
| warnings icon, in this guide . . . . .             | xxvii |
| warnings (pragma directive) . . . . .              | 246   |

--warnings\_affect\_exit\_code (compiler option) . . . . 122, 152  
 --warnings\_are\_errors (compiler option) . . . . . 152  
 wchar.h (library header file) . . . . . 219, 222  
 wchar.h, added C functionality . . . . . 224  
 wchar\_t (data type), adding support for in C . . . . . 155  
 wcstof, C99 extension . . . . . 224  
 wcstolb, C99 extension . . . . . 224  
 wctype.h (library header file) . . . . . 219  
 wctype.h, added C functionality . . . . . 224  
 web sites, recommended . . . . . xxvi  
 \_\_write (library function) . . . . . 59  
     customizing . . . . . 56

## X

XCFI\_COMMON (call frame information macro) . . . . . 89  
 XCFI\_NAMES (call frame information macro) . . . . . 89  
 XCHG.B (assembler instruction) . . . . . 201  
 XCHG.L (assembler instruction) . . . . . 202  
 XCHG.W (assembler instruction) . . . . . 202  
 XLINK errors  
     range error . . . . . 38  
     segment too long . . . . . 38  
 XLINK output files . . . . . 5  
 XLINK segment memory types . . . . . 28  
 xreportassert.c . . . . . 64

## Symbols

#include files, specifying . . . . . 120, 140  
 #warning message (preprocessor extension) . . . . . 215  
 -D (compiler option) . . . . . 131  
 -e (compiler option) . . . . . 137  
 -f (compiler option) . . . . . 139  
 -I (compiler option) . . . . . 140  
 -l (compiler option) . . . . . 75, 140  
 -O (compiler option) . . . . . 147  
 -o (compiler option) . . . . . 147–148  
 -r (compiler option) . . . . . 150

--align\_func (compiler option) . . . . . 130  
 --char\_is\_signed (compiler option) . . . . . 131  
 --code\_model (compiler option) . . . . . 131  
 --data\_model (compiler option) . . . . . 132  
 --debug (compiler option) . . . . . 132  
 --dependencies (compiler option) . . . . . 133  
 --diagnostics\_tables (compiler option) . . . . . 135  
 --diag\_error (compiler option) . . . . . 134  
 --diag\_remark (compiler option) . . . . . 134  
 --diag\_suppress (compiler option) . . . . . 134  
 --diag\_warning (compiler option) . . . . . 135  
 --discard\_unused\_publics (compiler option) . . . . . 136  
 --dlib\_config (compiler option) . . . . . 136  
 --double (compiler option) . . . . . 137  
 --ec++ (compiler option) . . . . . 137  
 --eec++ (compiler option) . . . . . 138  
 --enable\_multibytes (compiler option) . . . . . 138  
 --error\_limit (compiler option) . . . . . 138  
 --fp\_model (compiler option) . . . . . 139  
 --header\_context (compiler option) . . . . . 140  
 --library\_module (compiler option) . . . . . 141  
 --mfc (compiler option) . . . . . 141  
 --migration\_preprocessor\_extensions (compiler option) . . 142  
 --misrac (compiler option) . . . . . 129  
 --misrac\_verbose (compiler option) . . . . . 129  
 --misrac1998 (compiler option) . . . . . 129  
 --misrac2004 (compiler option) . . . . . 129  
 --module\_name (compiler option) . . . . . 142  
 --no\_code\_motion (compiler option) . . . . . 143  
 --no\_cross\_call (compiler option) . . . . . 143  
 --no\_cse (compiler option) . . . . . 143  
 --no\_inline (compiler option) . . . . . 144  
 --no\_path\_in\_file\_macros (compiler option) . . . . . 144  
 --no\_tbaa (compiler option) . . . . . 145  
 --no\_typedefs\_in\_diagnostics (compiler option) . . . . . 145  
 --no\_unroll (compiler option) . . . . . 146  
 --no\_warnings (compiler option) . . . . . 146  
 --no\_wrap\_diagnostics (compiler option) . . . . . 146  
 --omit\_types (compiler option) . . . . . 148

|                                                       |          |                                                        |          |
|-------------------------------------------------------|----------|--------------------------------------------------------|----------|
| --only_stdout (compiler option) . . . . .             | 148      | __enable_interrupt (intrinsic function) . . . . .      | 201      |
| --predef_macro (compiler option) . . . . .            | 149      | __exchange_byte (intrinsic function) . . . . .         | 201      |
| --preinclude (compiler option) . . . . .              | 149      | __exchange_long (intrinsic function) . . . . .         | 202      |
| --preprocess (compiler option) . . . . .              | 149      | __exchange_word (intrinsic function) . . . . .         | 202      |
| --remarks (compiler option) . . . . .                 | 150      | __exit (library function) . . . . .                    | 54       |
| --require_prototypes (compiler option) . . . . .      | 151      | __fast_interrupt (extended keyword) . . . . .          | 179      |
| --silent (compiler option) . . . . .                  | 151      | __FILE__ (predefined symbol) . . . . .                 | 212      |
| --strict_ansi (compiler option) . . . . .             | 151      | __FUNCTION__ (predefined symbol) . . . . .             | 172, 213 |
| --warnings_affect_exit_code (compiler option) . . . . | 122, 152 | __func__ (predefined symbol) . . . . .                 | 172, 213 |
| --warnings_are_errors (compiler option) . . . . .     | 152      | __gets, in stdio.h. . . . .                            | 224      |
| @ (operator)                                          |          | __get_DCR_register (intrinsic function) . . . . .      | 202      |
| placing at absolute address. . . . .                  | 104      | __get_DCT_register (intrinsic function) . . . . .      | 202      |
| placing in segments . . . . .                         | 105      | __get_DDA_register (intrinsic function) . . . . .      | 202      |
| _Exit (library function) . . . . .                    | 54       | __get_DDR_register (intrinsic function) . . . . .      | 202      |
| _exit (library function) . . . . .                    | 54       | __get_DMD_register (intrinsic function) . . . . .      | 203      |
| _Exit, C99 extension. . . . .                         | 224      | __get_DSA_register (intrinsic function) . . . . .      | 203      |
| _Pragma (predefined symbol) . . . . .                 | 215      | __get_DSR_register (intrinsic function) . . . . .      | 203      |
| __ALIGNOF__ (operator) . . . . .                      | 165      | __get_interrupt_level (intrinsic function) . . . . .   | 203      |
| __asm (language extension) . . . . .                  | 167      | __get_interrupt_state (intrinsic function) . . . . .   | 203      |
| __BASE_FILE__ (predefined symbol) . . . . .           | 212      | __get_interrupt_table (intrinsic function) . . . . .   | 204      |
| __break (intrinsic function) . . . . .                | 201      | __get_VCT_register (intrinsic function) . . . . .      | 204      |
| __BUILD_NUMBER__ (predefined symbol) . . . . .        | 212      | __IAR_SYSTEMS_ICC__ (predefined symbol) . . . . .      | 213      |
| __close (library function) . . . . .                  | 59       | __ICCR32C__ (predefined symbol) . . . . .              | 213      |
| __CODE_MODEL__ (predefined symbol) . . . . .          | 212      | __illegal_opcode (intrinsic function) . . . . .        | 204      |
| __code24 (extended keyword) . . . . .                 | 177      | __interrupt (extended keyword) . . . . .               | 23, 180  |
| __code32 (extended keyword) . . . . .                 | 178      | using in pragma directives . . . . .                   | 198      |
| __code32 (function pointer) . . . . .                 | 157      | __interrupt_on_overflow (intrinsic function) . . . . . | 204      |
| __core (runtime model attribute) . . . . .            | 68       | __intrinsic (extended keyword) . . . . .               | 180      |
| __cplusplus (predefined symbol) . . . . .             | 212      | __LINE__ (predefined symbol) . . . . .                 | 213      |
| __DATA_MODEL__ (predefined symbol) . . . . .          | 212      | __LITTLE_ENDIAN__ (predefined symbol) . . . . .        | 213      |
| __data16 (extended keyword) . . . . .                 | 178      | __load_context (intrinsic function) . . . . .          | 204      |
| __data24 (extended keyword) . . . . .                 | 179      | __low_level_init . . . . .                             | 53       |
| __data32 (data pointer) . . . . .                     | 158      | __low_level_init (intrinsic function) . . . . .        | 205      |
| __data32 (extended keyword) . . . . .                 | 179      | __low_level_init, customizing . . . . .                | 55       |
| __DATE__ (predefined symbol) . . . . .                | 212      | __lseek (library function) . . . . .                   | 59       |
| __delay_cycles (intrinsic function) . . . . .         | 201      | __monitor (extended keyword) . . . . .                 | 114, 180 |
| __disable_interrupt (intrinsic function) . . . . .    | 201      | __noreturn (extended keyword) . . . . .                | 181      |
| __double_size (runtime model attribute) . . . . .     | 69       | __no_init (extended keyword) . . . . .                 | 115, 181 |
| __embedded_cplusplus (predefined symbol) . . . . .    | 212      | __no_operation (intrinsic function) . . . . .          | 205      |

\_\_open (library function) . . . . . 59  
 \_\_packed (extended keyword) . . . . . 181  
 \_\_PRETTY\_FUNCTION\_\_ (predefined symbol) . . . . . 213  
 \_\_printf\_args (pragma directive) . . . . . 195, 246  
 \_\_program\_start (label) . . . . . 52  
 \_\_qsortbbl, C99 extension . . . . . 224  
 \_\_read (library function) . . . . . 59  
     customizing . . . . . 56  
 \_\_ReportAssert (library function) . . . . . 64  
 \_\_RMPA\_B (intrinsic function) . . . . . 205  
 \_\_RMPA\_L (intrinsic function) . . . . . 205  
 \_\_RMPA\_W (intrinsic function) . . . . . 205  
 \_\_root (extended keyword) . . . . . 182  
 \_\_ROUND (intrinsic function) . . . . . 101, 206  
 \_\_rt\_version (runtime model attribute) . . . . . 69  
 \_\_sbdata16 (extended keyword) . . . . . 182  
 \_\_sbdata24 (extended keyword) . . . . . 183  
 \_\_scanf\_args (pragma directive) . . . . . 196, 246  
 \_\_segment\_begin (extended operator) . . . . . 165  
 \_\_segment\_end (extended operators) . . . . . 165  
 \_\_set\_DCR\_register (intrinsic function) . . . . . 206  
 \_\_set\_DCT\_register (intrinsic function) . . . . . 206  
 \_\_set\_DDA\_register (intrinsic function) . . . . . 206  
 \_\_set\_DDR\_register (intrinsic function) . . . . . 206  
 \_\_set\_DMD\_register (intrinsic function) . . . . . 206  
 \_\_set\_DSA\_register (intrinsic function) . . . . . 207  
 \_\_set\_DSR\_register (intrinsic function) . . . . . 207  
 \_\_set\_interrupt\_level (intrinsic function) . . . . . 207  
 \_\_set\_interrupt\_state (intrinsic function) . . . . . 207  
 \_\_set\_interrupt\_table (intrinsic function) . . . . . 207  
 \_\_set\_VCT\_register (intrinsic function) . . . . . 207  
 \_\_SIN (intrinsic function) . . . . . 208  
 \_\_software\_interrupt (intrinsic function) . . . . . 208  
 \_\_SOUT (intrinsic function) . . . . . 208  
 \_\_STDC\_VERSION\_\_ (predefined symbol) . . . . . 213  
 \_\_STDC\_\_ (predefined symbol) . . . . . 213  
 \_\_STOP (intrinsic function) . . . . . 208  
 \_\_store\_context (intrinsic function) . . . . . 208  
 \_\_SUBVERSION\_\_ (predefined symbol) . . . . . 214

\_\_task (extended keyword) . . . . . 183  
 \_\_TIME\_\_ (predefined symbol) . . . . . 214  
 \_\_ungetchar, in stdio.h . . . . . 224  
 \_\_VA\_ARGS\_\_ (preprocessor extension) . . . . . 215  
 \_\_VER\_\_ (predefined symbol) . . . . . 214  
 \_\_wait\_for\_interrupt (intrinsic function) . . . . . 209  
 \_\_write (library function) . . . . . 59  
     customizing . . . . . 56  
 \_\_write\_array, in stdio.h . . . . . 224  
 \_\_write\_buffered (library function) . . . . . 67

## Numerics

32-bits (floating-point format) . . . . . 156  
 64-bit data types, avoiding . . . . . 99  
 64-bits (floating-point format) . . . . . 157