COPYRIGHT NOTICE
Copyright © 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. RX is a trademark of Renesas Technology Corporation.

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

EDITION NOTICE
First edition: May 2009
Part number: CRX-1

This guide applies to version 1.x of IAR Embedded Workbench® for RX.
Internal reference: R8, 5.4, IJOA.
Brief contents

Tables ..................................................................................................................... xix
Preface ................................................................................................................... xxi

Part 1. Using the compiler ................................................................................... 1
Getting started .................................................................................................... 3
Data storage ...................................................................................................... 11
Functions ............................................................................................................... 21
Placing code and data ........................................................................................ 29
The DLIB runtime environment ................................................................. 43
Assembler language interface .................................................................... 73
Using C++ ............................................................................................................ 91
Efficient coding for embedded applications ............................................ 101

Part 2. Reference information .......................................................................... 119
External interface details ................................................................................ 121
Compiler options .............................................................................................. 127
Data representation .......................................................................................... 155
Compiler extensions ......................................................................................... 167
Extended keywords .......................................................................................... 177
Pragma directives ............................................................................................... 189
Intrinsic functions ............................................................................................ 203
The preprocessor ............................................................................................... 209
Library functions .............................................................................................. 215
Segment reference ................................................................. 223
Implementation-defined behavior ............................................. 235
Index ...................................................................................... 247
Contents

Tables ..................................................................................................................... xix
Preface ................................................................................................................... xxi

Who should read this guide ..............................................................xxi
How to use this guide .........................................................................xxi
What this guide contains .................................................................xxii
Other documentation ........................................................................xxiii
Further reading ..................................................................................xxiii
Document conventions ........................................................................xxiv
Typographic conventions ................................................................xxiv
Naming conventions ........................................................................xxv

Part 1. Using the compiler ............................................................ 1
Getting started ............................................................................................. 3

IAR language overview ........................................................................3
Supported RX devices ........................................................................4
Building applications—an overview .................................................. 4
Compiling ................................................................................................. 4
Linking ........................................................................................................ 5

Basic settings for project configuration ......................................... 5
Data model ............................................................................................... 6
Code model .............................................................................................. 6
Size of double floating-point type ....................................................... 6
Byte order ................................................................................................. 7
Optimization for speed and size ............................................................ 7
Runtime environment ........................................................................... 7

Special support for embedded systems ........................................... 9
Extended keywords ................................................................................. 9
Pragma directives .................................................................................... 9
Predefined symbols ............................................................................. 9
Special function types ...................................................................... 9
## Data storage

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing low-level features</td>
<td>10</td>
</tr>
<tr>
<td>Data storage</td>
<td>11</td>
</tr>
<tr>
<td>Introduction</td>
<td>11</td>
</tr>
<tr>
<td>Different ways to store data</td>
<td>11</td>
</tr>
<tr>
<td>Data models</td>
<td>12</td>
</tr>
<tr>
<td>Specifying a data model</td>
<td>12</td>
</tr>
<tr>
<td>Memory types</td>
<td>13</td>
</tr>
<tr>
<td>Data16</td>
<td>13</td>
</tr>
<tr>
<td>Data24</td>
<td>14</td>
</tr>
<tr>
<td>Data32</td>
<td>14</td>
</tr>
<tr>
<td>Using data memory attributes</td>
<td>14</td>
</tr>
<tr>
<td>Structures and memory types</td>
<td>15</td>
</tr>
<tr>
<td>More examples</td>
<td>16</td>
</tr>
<tr>
<td>C++ and memory types</td>
<td>17</td>
</tr>
<tr>
<td>Auto variables—on the stack</td>
<td>17</td>
</tr>
<tr>
<td>The stack</td>
<td>18</td>
</tr>
<tr>
<td>Dynamic memory on the heap</td>
<td>19</td>
</tr>
<tr>
<td>Functions</td>
<td>21</td>
</tr>
<tr>
<td>Function-related extensions</td>
<td>21</td>
</tr>
<tr>
<td>Code models and memory attributes for function storage</td>
<td>21</td>
</tr>
<tr>
<td>Using function memory attributes</td>
<td>22</td>
</tr>
<tr>
<td>Primitives for interrupts, concurrency, and OS-related programming</td>
<td>23</td>
</tr>
<tr>
<td>Interrupt functions</td>
<td>23</td>
</tr>
<tr>
<td>Fast interrupt functions</td>
<td>24</td>
</tr>
<tr>
<td>Monitor functions</td>
<td>24</td>
</tr>
<tr>
<td>C++ and special function types</td>
<td>27</td>
</tr>
<tr>
<td>Placing code and data</td>
<td>29</td>
</tr>
<tr>
<td>Segments and memory</td>
<td>29</td>
</tr>
<tr>
<td>What is a segment?</td>
<td>29</td>
</tr>
<tr>
<td>Placing segments in memory</td>
<td>30</td>
</tr>
<tr>
<td>Customizing the linker command file</td>
<td>31</td>
</tr>
</tbody>
</table>
## Data segments
- Static memory segments .......................................................... 33
- The stacks .................................................................................. 36
- The heap .................................................................................... 38
- Located data .............................................................................. 39

## Code segments
- Startup code ............................................................................. 39
- Normal code .............................................................................. 39
- Interrupt vectors ...................................................................... 40

## C++ dynamic initialization
- Verifying the linked result of code and data placement ......... 40
  - Segment too long errors and range errors ......................... 40
  - Linker map file ...................................................................... 41

## The DLIB runtime environment ............................................. 43
- Introduction to the runtime environment ................................. 43
  - Runtime environment functionality .................................... 43
  - Library selection .................................................................... 44
  - Situations that require library building ............................... 45
  - Library configurations .......................................................... 45
  - Debug support in the runtime library .................................. 46
- Using a prebuilt library ............................................................ 46
  - Customizing a prebuilt library without rebuilding ............. 48
- Choosing formatters for printf and scanf .............................. 49
  - Choosing printf formatter .................................................... 49
  - Choosing scanf formatter .................................................... 50
- Overriding library modules ..................................................... 51
- Building and using a customized library ................................. 52
  - Setting up a library project ................................................. 52
  - Modifying the library functionality ................................... 53
  - Using a customized library .................................................. 53
- System startup and termination .............................................. 54
  - System startup ..................................................................... 54
  - System termination .............................................................. 55
Customizing system initialization ................................................... 56
  __low_level_init ................................................................. 57
Modifying the file cstartup.s54 .................................................... 57
Standard streams for input and output ........................................ 57
  Implementing low-level character input and output ..................... 58
Configuration symbols for printf and scanf .............................. 60
  Customizing formatting capabilities ........................................ 61
File input and output ................................................................... 61
Locale ...................................................................................... 62
  Locale support in prebuilt libraries ........................................... 62
  Customizing the locale support .............................................. 62
  Changing locales at runtime ................................................. 63
Environment interaction ............................................................. 64
Signal and raise ........................................................................ 65
Time ...................................................................................... 65
Strtod ..................................................................................... 65
Assert ..................................................................................... 66
Hardware support ..................................................................... 66
C-SPY runtime interface ........................................................... 66
  Low-level debugger runtime interface ....................................... 67
  The debugger terminal I/O window ......................................... 67
Checking module consistency ..................................................... 68
  Runtime model attributes ..................................................... 68
  Using runtime model attributes .......................................... 69
  Predefined runtime attributes ............................................. 70
  User-defined runtime model attributes ................................ 71
Assembler language interface ..................................................... 73
Mixing C and assembler .......................................................... 73
  Intrinsic functions ............................................................. 73
  Mixing C and assembler modules ........................................ 74
  Inline assembler ................................................................... 75
Calling assembler routines from C .......................................... 76
  Creating skeleton code ....................................................... 76
Compiling the code ................................................................. 77

**Calling assembler routines from C++** ........................... 78

**Calling convention** .......................................................... 79
  Function declarations ......................................................... 80
  Using C linkage in C++ source code ..................................... 80
  Preserved versus scratch registers ..................................... 81
  Function entrance ............................................................ 81
  Function exit .................................................................... 83
  Restrictions for special function types ............................... 84
  Examples .......................................................................... 84
  Function directives .......................................................... 86

**Calling functions** ............................................................ 86
  Assembler instructions used for calling functions ................ 87

**Call frame information** .................................................... 87
  CFI directives .................................................................... 87
  Creating assembler source with CFI support ..................... 88

**Using C++** ...................................................................... 91

**Overview** ....................................................................... 91
  Standard Embedded C++ .................................................. 91
  Extended Embedded C++ .................................................. 92
  Enabling C++ support ...................................................... 92

**Feature descriptions** ....................................................... 93
  Classes ............................................................................ 93
  Functions ......................................................................... 96
  Templates ......................................................................... 96
  Variants of casts ............................................................. 97
  Mutable ............................................................................ 97
  Namespace ........................................................................ 97
  The STD namespace ........................................................ 97
  Using interrupts and EC++ destructors .............................. 97

**C++ language extensions** .................................................. 98
Efficient coding for embedded applications ........................................ 101

Selecting data types ................................................................. 101
Using efficient data types ......................................................... 101
Floating-point types .............................................................. 102
Alignment of elements in a structure ........................................ 103
Anonymous structs and unions .................................................. 103

Controlling data and function placement in memory .................. 105
Data placement at an absolute location .................................... 106
Data and function placement in segments .................................. 107

Controlling compiler optimizations .......................................... 108
Scope for performed optimizations ......................................... 109
Optimization levels ............................................................... 109
Speed versus size ................................................................... 110
Fine-tuning enabled transformations ....................................... 110

Facilitating good code generation ............................................ 113
Writing optimization-friendly code ......................................... 113
Saving stack space and RAM memory ..................................... 114
Aligning the function entry point ............................................ 114
Register locking ..................................................................... 114
Function prototypes ............................................................. 115
Integer types and bit negation ................................................ 116
Protecting simultaneously accessed variables ......................... 116
Accessing special function registers ...................................... 117
Non-initialized variables ....................................................... 118

Part 2. Reference information .................................................. 119

External interface details .......................................................... 121

Invocation syntax ................................................................. 121
Compiler invocation syntax .................................................... 121
Passing options ...................................................................... 121
Environment variables .......................................................... 122

Include file search procedure ................................................ 122
Compiler output .......................................................... 123
Diagnostics ........................................................................ 124
  Message format .............................................................. 125
  Severity levels ............................................................. 125
  Setting the severity level .............................................. 126
  Internal error ............................................................. 126
Compiler options ........................................................... 127
  Options syntax ............................................................. 127
    Types of options ......................................................... 127
    Rules for specifying parameters .................................. 127
Summary of compiler options ......................................... 130
Descriptions of options ................................................ 132
  --align_func .............................................................. 132
  --char_is_signed ...................................................... 133
  --code_model .......................................................... 133
  --core ................................................................. 133
  -D .......................................................... 134
  --data_model .......................................................... 134
  --debug, -r ......................................................... 135
  --dependencies ....................................................... 135
  --diag_error .......................................................... 136
  --diag_remark ......................................................... 136
  --diag_suppress ....................................................... 137
  --diag_warning ....................................................... 137
  --diagnostics_tables ................................................ 137
  --discard_unused_publics ........................................ 138
  --dlib_config ........................................................ 138
  --double ............................................................. 139
  -e ................................................................. 139
  --ec++ ............................................................. 140
  --eec++ ............................................................ 140
  --enable_multibytes ............................................... 140
  --endian .............................................................. 140
--error_limit ...................................................................................... 141
-l .......................................................................................................... 141
--header_context ............................................................................... 142
-l .......................................................................................................... 142
-l .......................................................................................................... 142
--library_module .............................................................................. 143
--lock ............................................................................................... 143
--mfc ............................................................................................... 144
--module_name ................................................................................ 144
--no_clustering ............................................................................... 145
--no_code_motion ............................................................................. 145
--no_cross_call ............................................................................... 145
--no_cse ............................................................................................ 146
--no_inline ........................................................................................ 146
--no_path_in_file_macros ................................................................. 147
--no_tbaa ......................................................................................... 147
--no_typedefs_in_diagnostics ......................................................... 147
--no_unroll ....................................................................................... 148
--no_warnings ................................................................................ 148
--no_wrap_diagnostics ................................................................. 149
-O ...................................................................................................... 149
-o, --output ....................................................................................... 150
--omit_types .................................................................................... 150
--only_stdout .................................................................................. 150
--output, -o ..................................................................................... 151
--predef Macros ................................................................................ 151
--preinclude ................................................................................... 151
--preprocess .................................................................................... 152
--public_equ ..................................................................................... 152
-r, --debug ...................................................................................... 152
--remarks ........................................................................................ 153
--require_prototypes ...................................................................... 153
--silent ............................................................................................. 153
--strict_ansi ................................................................................... 154
Descriptions of extended keywords ............................................. 181
__code24 .......................................................................................... 181
__code32 .......................................................................................... 182
__data16 ........................................................................................... 182
__data24 ........................................................................................... 183
__data32 ........................................................................................... 183
__fast_interrupt ................................................................................ 184
__interrupt ........................................................................................ 184
__intrinsic ........................................................................................ 184
__monitor ......................................................................................... 184
__no_init .......................................................................................... 185
__noreturn ........................................................................................ 185
__packed .......................................................................................... 185
__root ............................................................................................... 186
__task ............................................................................................... 186
Pragma directives .............................................................................. 189

Summary of pragma directives ...................................................... 189
Descriptions of pragma directives ................................................ 190
bitfields ............................................................................................. 190
constseg ............................................................................................ 191
data_alignment .................................................................................. 191
dataseg .............................................................................................. 192
diag_default ...................................................................................... 192
diag_error ........................................................................................ 193
diag_remark ...................................................................................... 193
diag_suppress ................................................................................... 193
diag_warning .................................................................................... 194
include_alias ..................................................................................... 194
inline ................................................................................................. 195
language ............................................................................................ 195
location ............................................................................................. 196
message ............................................................................................ 196
object_attribute ................................................................................. 197
Intrinsic functions

Summary of intrinsic functions .................................................. 203

Descriptions of intrinsic functions ................................................. 204

__break ............................................................................................ 204
__delay_cycles ................................................................................. 204
__disable_interrupt .......................................................................... 204
__enable_interrupt ........................................................................... 204
__exchange ...................................................................................... 205
__get_FINTV_register ..................................................................... 205
__get_interrupt_level ....................................................................... 205
__get_interrupt_state ....................................................................... 205
__get_interrupt_table ....................................................................... 206
__illegal_opcode .............................................................................. 206
__no_operation ................................................................................ 206
__RMPA_B ..................................................................................... 206
__RMPA_L ...................................................................................... 206
__RMPA_W .................................................................................... 207
__ROUND ....................................................................................... 207
__set_FINTV_register ..................................................................... 207
__set_interrupt_level ....................................................................... 207
__set_interrupt_state ........................................................................ 207
__set_interrupt_table ....................................................................... 208
__software_interrupt ........................................................................ 208
__wait_for_interrupt ........................................................................ 208
The preprocessor ........................................................................................... 209

Overview of the preprocessor ............................................................... 209

Descriptions of predefined preprocessor symbols ..................... 210

Descriptions of miscellaneous preprocessor extensions .......... 212

- NDEBUG ......................................................................................... 212
- _Pragma() ......................................................................................... 213
- #warning message ............................................................................ 213
- __VA_ARGS__ ............................................................................... 213

Library functions ............................................................................................. 215

Introduction ............................................................................................ 215

- Header files .................................................................................. 215
- Library object files ........................................................................ 215
- Reentrancy ....................................................................................... 216

IAR DLIB Library .................................................................................. 216

- C header files ................................................................................... 217
- C++ header files ............................................................................... 218
- Library functions as intrinsic functions ........................................... 220
- Added C functionality ..................................................................... 220

Segment reference ......................................................................................... 223

Summary of segments .............................................................................. 223

Descriptions of segments ...................................................................... 224

- CHECKSUM ................................................................................... 225
- CODE24 ........................................................................................... 225
- CODE32 ........................................................................................... 225
- CSTART .......................................................................................... 225
- DATA16_AC .................................................................................. 226
- DATA16_AN .................................................................................. 226
- DATA16_C ..................................................................................... 226
- DATA16_I ...................................................................................... 226
- DATA16_ID ................................................................................... 226
- DATA16_N ..................................................................................... 227
- DATA16_Z ...................................................................................... 227
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA24_AC</td>
<td>228</td>
</tr>
<tr>
<td>DATA24_AN</td>
<td>228</td>
</tr>
<tr>
<td>DATA24_C</td>
<td>228</td>
</tr>
<tr>
<td>DATA24_I</td>
<td>228</td>
</tr>
<tr>
<td>DATA24_ID</td>
<td>229</td>
</tr>
<tr>
<td>DATA24_N</td>
<td>229</td>
</tr>
<tr>
<td>DATA24_Z</td>
<td>229</td>
</tr>
<tr>
<td>DATA32_AC</td>
<td>230</td>
</tr>
<tr>
<td>DATA32_AN</td>
<td>230</td>
</tr>
<tr>
<td>DATA32_C</td>
<td>230</td>
</tr>
<tr>
<td>DATA32_I</td>
<td>230</td>
</tr>
<tr>
<td>DATA32_ID</td>
<td>231</td>
</tr>
<tr>
<td>DATA32_N</td>
<td>231</td>
</tr>
<tr>
<td>DATA32_Z</td>
<td>231</td>
</tr>
<tr>
<td>DIFUNCT</td>
<td>232</td>
</tr>
<tr>
<td>HEAP</td>
<td>232</td>
</tr>
<tr>
<td>INTVEC</td>
<td>232</td>
</tr>
<tr>
<td>ISTACK</td>
<td>232</td>
</tr>
<tr>
<td>NMIVEC</td>
<td>233</td>
</tr>
<tr>
<td>SWITCH</td>
<td>233</td>
</tr>
<tr>
<td>USTACK</td>
<td>233</td>
</tr>
</tbody>
</table>

**Implementation-defined behavior** .......................................................... 235

**Descriptions of implementation-defined behavior** ............................... 235

- Translation ....................................................................................... 235
- Environment ...................................................................................... 236
- Identifiers ......................................................................................... 236
- Characters .......................................................................................... 236
- Integers ............................................................................................. 238
- Floating point ................................................................................... 238
- Arrays and pointers .......................................................................... 239
- Registers ............................................................................................. 239
- Structures, unions, enumerations, and bitfields ............................... 239
- Qualifiers .......................................................................................... 240
Tables

1: Typographic conventions used in this guide ........................................... xxiv
2: Naming conventions used in this guide ................................................... xxv
3: Command line options for specifying library and dependency files .......... 8
4: Data model characteristics ....................................................................... 12
5: Memory types and their corresponding memory attributes ................... 14
6: Code models ............................................................................................ 22
7: Function memory attributes ..................................................................... 22
8: XLINK segment memory types ................................................................. 30
9: Memory layout of a target system (example) ........................................... 31
10: Memory types with corresponding segment groups ............................... 34
11: Segment name suffixes .......................................................................... 34
12: Library configurations ............................................................................ 45
13: Levels of debugging support in runtime libraries ................................... 46
14: Prebuilt libraries .................................................................................... 47
15: Customizable items ................................................................................ 48
16: Formatters for printf ............................................................................ 49
17: Formatters for scanf ............................................................................. 50
18: Descriptions of printf configuration symbols ........................................ 60
19: Descriptions of scanf configuration symbols ......................................... 60
20: Low-level I/O files ................................................................................ 61
21: Functions with special meanings when linked with debug info ............... 66
22: Example of runtime model attributes ..................................................... 69
23: Predefined runtime model attributes ..................................................... 70
24: Registers used for passing parameters .................................................... 82
25: Registers used for returning values ......................................................... 83
26: Call frame information resources defined in a names block .................. 88
27: Compiler optimization levels .................................................................. 109
28: Compiler environment variables ............................................................ 122
29: Error return codes ................................................................................ 124
30: Compiler options summary .................................................................... 130
31: Integer types .......................................................................................... 156
32: Floating-point types ................................................................. 158
33: Extended keywords summary .................................................. 181
34:Pragma directives summary ...................................................... 189
35: Intrinsic functions summary ..................................................... 203
36:Predefined symbols ................................................................. 210
37: Traditional standard C header files—DLIB ............................. 217
38: Embedded C++ header files ..................................................... 218
39: Additional Embedded C++ header files—DLIB ...................... 218
40: Standard template library header files .................................... 219
41: New standard C header files—DLIB ........................................ 219
42: Segment summary ................................................................. 223
43: Message returned by strerror()—IAR DLIB library ............... 245
Preface

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

- The architecture and instruction set of the RX microcontroller. Refer to the documentation from Renesas for information about the RX microcontroller
- 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 RX, 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 RX-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 RX-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 RX microcontroller 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 RX IAR Assembler, refer to the *IAR Assembler Reference Guide for RX*

● 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


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:

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 rx\doc, the full path to the location is assumed, for example c:\Program Files\IAR Systems\Embedded Workbench 5.1\rx\doc.

**TYPOGRAPHIC CONVENTIONS**

This guide uses the following typographic conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Used for</th>
</tr>
</thead>
</table>
| computer  | • Source code examples and file paths.  
|           | • Text on the command line.  
|           | • Binary, hexadecimal, and octal numbers.                                 |
| parameter | A placeholder for an actual value used as a parameter, for example filename.h where filename represents the name of the file. |
| [option]  | An optional part of a command.                                           |
| a|b|c       | Alternatives in a command.                                               |

*Table 1: Typographic conventions used in this guide*
NAMING CONVENTIONS

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

<table>
<thead>
<tr>
<th>Brand name</th>
<th>Generic term</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR Embedded Workbench® for RX</td>
<td>IAR Embedded Workbench®</td>
</tr>
<tr>
<td>IAR Embedded Workbench® IDE for RX</td>
<td>the IDE</td>
</tr>
<tr>
<td>IAR C-SPY® Debugger for RX</td>
<td>C-SPY, the debugger</td>
</tr>
<tr>
<td>IAR C-SPY® Simulator</td>
<td>the simulator</td>
</tr>
<tr>
<td>IAR C/C++ Compiler™ for RX</td>
<td>the compiler</td>
</tr>
<tr>
<td>IAR Assembler™ for RX</td>
<td>the assembler</td>
</tr>
<tr>
<td>IAR XLINK™ Linker</td>
<td>XLINK, the linker</td>
</tr>
<tr>
<td>IAR XAR Library builder™</td>
<td>the library builder</td>
</tr>
<tr>
<td>IAR XLIB Librarian™</td>
<td>the librarian</td>
</tr>
<tr>
<td>IAR DLIB Library™</td>
<td>the DLIB library</td>
</tr>
</tbody>
</table>

Table 2: Naming conventions used in this guide
Part 1. Using the compiler

This part of the IAR C/C++ Compiler Reference Guide for RX 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 RX microcontroller. 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 RX:

- C, the most widely used high-level programming language in the embedded systems industry. Using the IAR C/C++ Compiler for RX, 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 IAR Assembler Reference Guide for RX.

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

**Supported RX devices**

The IAR C/C++ Compiler for RX supports all devices based on the standard Renesas RX600 microcomputer series. The following extensions are also supported:

- 32-bit multiplier and divider
- Single-precision hardware floating-point unit (FPU).

**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.r54` using the default settings:

```
iccrx 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:

```command
xlink myfile.r54 myfile2.r54 -s __program_start -f lnkrx.xcl dlrxfdslf.r54 -o aout.a54 -r
```

In this example, `myfile.r54` and `myfile2.r54` are object files, `lnkrx.xcl` is the linker command file, and `dlrxfdslf.r54` 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 RX 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
- Size of double floating-point type
- Byte order
Basic settings for project configuration

- 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 RX microcontroller 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, it is possible to override the default access method for each individual variable. These data models are supported:

- The *Near* data model can access the highest and lowest 32 Kbytes of memory
- The *Far* data model can access the highest and lowest 8 Mbytes of memory
- The *Huge* data model can access the entire 32-bit address area.

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. These code models are available:

- The *Far* code model can access the lowest and highest 8 Mbytes of memory
- The *Huge* code model can access the entire 32-bit address area.

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 IEEE 754 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.
BYTE ORDER
For data access, the RX architecture allows a choice between the big- and little-endian byte order, see --endian, page 140.

Note: In big-endian mode, the chip operates on four-byte chunks. If you change the byte order between segment parts in a big-endian application, each segment part must begin on a 4-byte aligned address, or linking will fail with an Alignment Error. See the IAR Assembler Reference Guide for RX for more information about the assembler directives that toggle between code and data sections in linker segments.

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

The runtime library you choose can be one of the prebuilt libraries, or a library that you customized and built yourself. The IDE provides library project templates that you can use for building your own library versions. 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 environments, 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 45, 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:

<table>
<thead>
<tr>
<th>Command line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-I rx\inc</code></td>
<td>Specifies the include path to device-specific I/O definition files.</td>
</tr>
<tr>
<td><code>libraryfile.r54</code></td>
<td>Specifies the library object file.</td>
</tr>
<tr>
<td><code>--dlib_config C:\...\configfile.h</code></td>
<td>Specifies the library configuration file.</td>
</tr>
</tbody>
</table>

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

For a list of all prebuilt library object files for the IAR DLIB Library, see Table 14, *Prebuilt libraries*, page 47. 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 49.
- The size of the stacks and the heap, see *The stacks*, page 36, and *The heap*, page 38, respectively.
Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the RX microcontroller.

**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 139 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 RX microcontroller 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 23.
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 73.

Special support for embedded systems
Data storage

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

Introduction

The RX microcontroller has one continuous memory space for both code and data, ranging from 0x00000000 to 0xFFFFFFFF. Different types of memory can be placed in the memory range. A typical application will have ROM memory in the upper address interval, and RAM in the lower address interval.

Both code and data can be efficiently read. Physically, data and code reside on different memory buses, but the address spaces are disjoint.

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
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, and constant literals.

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

<table>
<thead>
<tr>
<th>Data model name</th>
<th>Default memory attribute</th>
<th>Pointer attribute</th>
<th>Placement of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Near</td>
<td>__data16</td>
<td>__data32</td>
<td>Low 32 Kbytes or high 32 Kbytes</td>
</tr>
<tr>
<td>Far (default)</td>
<td>__data24</td>
<td>__data32</td>
<td>Low 8 Mbytes or high 8 Mbytes</td>
</tr>
<tr>
<td>Huge</td>
<td>__data32</td>
<td>__data32</td>
<td>The entire 4 Gbytes of memory</td>
</tr>
</tbody>
</table>

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 134.

The impact of the different data models on code size depends on the amount of data with static duration. There is no principal difference in the generated code. On higher...
optimization levels the difference is even smaller, because of the global clustering optimization.

The RX microcontroller has no direct addressing mode. This means that addresses of static objects must be loaded into a register before the data can be read from memory. The size of these address loads will increase if you change to a larger data model. However, on high optimization levels, the compiler will use a base address to all objects with static duration data in the module, and use relative addressing to access them. For this reason, the size of the generated code does not depend very much on your choice of data model, but you should always use the smallest data model that you need.

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.

**DATA16**

The data16 memory consists of the highest and the lowest 32 Kbytes of data memory. In hexadecimal notation, this is the address ranges \(0x00000000-0x00007FFF\) 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. If you use objects of this type, the code generated by the compiler to access them becomes slightly smaller. This means a smaller footprint for the application, and faster execution at runtime.
DATA24

The data24 memory consists of the highest and the lowest 8 Mbytes of data memory. In hexadecimal notation, this is the address ranges 0x00000000-0x007FFFFF and 0xFF800000-0xFFFFFFFF.

A data24 object can only be placed in data24 memory, and the size of such an object is limited to 8 Mbytes-1.

DATA32

Data32 objects can be placed anywhere in the data memory space. Also, unlike the other memory types, there is no limitation on the size of the objects that can be placed in this memory type.

The data32 memory type uses 4-byte addresses, which can make the code slightly larger.

The compiler will optimize direct accesses (using literal addresses) so that the size penalty for using different memory types becomes smaller.

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.

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

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Keyword</th>
<th>Address range</th>
<th>Default in data model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data16</td>
<td>__data16</td>
<td>0x00000000-0x00007FFF and 0xFFFF8000-0xFFFFFFFF</td>
<td>Near</td>
</tr>
<tr>
<td>Data24</td>
<td>__data24</td>
<td>0x00000000-0x007FFFFF and 0xFFFF8000-0xFFFFFFFF</td>
<td>Far</td>
</tr>
<tr>
<td>Data32</td>
<td>__data32</td>
<td>0x00000000-0xFFFFFFFF</td>
<td>Huge</td>
</tr>
</tbody>
</table>

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 139 for additional information.
For reference information about each keyword, see *Descriptions of extended keywords*, page 181.

**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 177.

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:

```c
__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:

```c
/* Defines via a typedef */
typedef char __data32r Byte;
typedef Byte *BytePtr;
Byte AByte;
BytePtr ABytePointer;

/* Defines directly */
__data32r char AByte;
char __data32r *ABytePointer;
```

**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 \texttt{gamma} is a structure placed in data24 memory.

\begin{verbatim}
struct MyStruct
{
  int mAlpha;
  int mBeta;
};
__data24 struct MyStruct Gamma;
\end{verbatim}

This declaration is incorrect:

\begin{verbatim}
struct MyStruct
{
  int mAlpha;
  __data24 int mBeta;
};
\end{verbatim}

**MORE EXAMPLES**

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in data16 memory is declared. The function returns a pointer to an integer in data24 memory. 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.

\begin{verbatim}
int MyA;  \hspace{1cm} \text{A variable defined in default memory determined by the data model in use.}
int __data16 MyB; \hspace{1cm} \text{A variable in data16 memory.}
__data24 int MyC; \hspace{1cm} \text{A variable in data24 memory.}
int * MyD; \hspace{1cm} \text{A pointer stored in default memory. The pointer points to an integer anywhere in memory.}
int __data16 * MyE; \hspace{1cm} \text{A pointer stored in default memory. The pointer points to an integer in data16 memory.}
int __data16 * __data24 MyF; \hspace{1cm} \text{A pointer stored in data24 memory pointing to an integer stored in data16 memory.}
int __data24 * MyFunction(
  int __data16 *); \hspace{1cm} \text{A declaration of a function that takes a parameter which is a pointer to an integer stored in data16 memory. The function returns a pointer to an integer stored in data24 memory.}
\end{verbatim}
**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.

**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.

```cpp
// A class definition (may be placed in a header file)
class MyClass
{
public:
    int mAlpha;
    int mBeta;

    __data16 static int mGamma;
};

// Needed definitions (should be placed in a source file)
__data16 int MyClass::mGamma;

// An object of class type MyClass
__data24 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.

```c
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.
Dynamic memory 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 Facilitating good code generation, page 113. 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 in a default part of memory, or in other words, use a default size of the function address. Technically, the code models control the default memory range for storing the function, which implies a 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:

<table>
<thead>
<tr>
<th>Code model</th>
<th>Default address range for placing functions and constant data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Far (default)</td>
<td>0x00000000–0x007FFFFF 0xFF800000–0xFFFFFFFF</td>
</tr>
<tr>
<td>Huge</td>
<td>0x00000000–0xFFFFFFFF</td>
</tr>
</tbody>
</table>

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 133.

If the device you are using has the code memory split up in several parts, and the maximum distance between the parts is greater than 8 Mbytes, you must use the Huge code model if you want to be able to use the entire code memory without using function memory attributes.

Alternatively, you can use the Far code model (the compiler default) and use the memory attribute __code32 on some functions to place them in remote address areas. Make sure you declare these functions __code32 in all occurrences so that functions placed by default (in code24 memory) can call them.

**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:

<table>
<thead>
<tr>
<th>Function memory attribute</th>
<th>Address range</th>
<th>Default in code model</th>
</tr>
</thead>
<tbody>
<tr>
<td>__code24</td>
<td>0x00000000–0x007FFFFF 0xFF800000–0xFFFFFFFF</td>
<td>Far</td>
</tr>
<tr>
<td>__code32</td>
<td>0x00000000–0xFFFFFFFF</td>
<td>Huge</td>
</tr>
</tbody>
</table>

Table 7: Function memory attributes

All pointers are 32 bits.

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For details about the restrictions, see Casting, page 160.

For detailed syntax information and for detailed information about each attribute, see the chapter Extended keywords.
Primitives for interrupts, concurrency, and OS-related programming

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

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

Interrupt Functions

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

In general, when an interrupt occurs in the code, the microcontroller 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 RX microcontroller 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 RX microcontroller documentation from the chip manufacturer. The `INTB` (interrupt table) register points to the start of the interrupt vector. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors. For the RX microcontroller, the placement of the interrupt table can be controlled at link time, by assigning the address space of the `INTVEC` segment.

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:

```c
#pragma vector = TMRA0 /* Symbol defined in I/O header file */
__interrupt void MyInterruptRoutine(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 RX microcontroller documentation for more information about the interrupt vector table.

**FAST INTERRUPT FUNCTIONS**

A fast interrupt function is very fast and has the highest priority. A fast interrupt uses the FREIT return mechanism and the FINTV register as a vector. Use the intrinsic function __set_FINTV_register to initialize this vector register, see __set_FINTV_register, page 207.

**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 184.

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

**Example of implementing a semaphore in C**

In the following example, a binary semaphore—that is, a mutex—is implemented using one static variable and two monitor functions. A monitor function works like a critical region, that is no interrupt can occur and the process itself cannot be swapped out. A semaphore can be locked by one process, and is used for preventing processes from simultaneously using resources that can only be used by one process at a time, for example a USART. The __monitor keyword assures that the lock operation is atomic; in other words it cannot be interrupted.

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

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

__monitor int TryGetLock(void)
{
    if (sTheLock == 0)
Functions

```c
/* Success, nobody has the lock. */

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

    return 0;
}
}

/* Function to unlock the lock.
   * It is only callable by one that has the lock.
   */
__monitor void ReleaseLock(void)
{
    sTheLock = 0;
}

/* Function to take the lock. It will wait until it gets it. */
void GetLock(void)
{
    while (!TryGetLock())
    {
        /* Normally a sleep instruction is used here. */
    }
}

/* An example of using the semaphore. */
void MyProgram(void)
{
    GetLock();
    /* ... Do something ... */

    ReleaseLock();
}
```
Example of implementing a semaphore in C++

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

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

```cpp
#include <intrinsics.h>

/* Class for controlling critical blocks. */
class Mutex
{
public:
    Mutex();
    ~Mutex();
private:
    __istate_t mState;
};

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

        // Enter a critical block.
        Mutex m;

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

private:
  static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;
extern void DoStuff();

void MyMain()
{
  static long nextStop = 100;
  if (Tick::GetTick() >= nextStop)
  {
    nextStop += 100;
    DoStuff();
  }
}

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, one restriction applies:

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

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

The compiler has several predefined segments for different purposes. Each segment 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:

<table>
<thead>
<tr>
<th>Segment memory type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>For executable code</td>
</tr>
<tr>
<td>CONST</td>
<td>For data placed in ROM</td>
</tr>
<tr>
<td>DATA</td>
<td>For data placed in RAM</td>
</tr>
</tbody>
</table>

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 the 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:

<table>
<thead>
<tr>
<th>Range</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000–0x1FFF</td>
<td>RAM</td>
</tr>
<tr>
<td>0x2000–0xCFPP</td>
<td>ROM</td>
</tr>
<tr>
<td>0x10000–0x11FFF</td>
<td>RAM</td>
</tr>
<tr>
<td>0x20000–0x3FFFF</td>
<td>ROM</td>
</tr>
</tbody>
</table>

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:
  -crx
  This specifies your target microcontroller.

- 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.

- For big-endian applications, the XLINK options -hc and -H must be used. The -hc option inverts the byte order for data to make it big-endian, and the -H option fills all gaps between segment parts that the linker creates with a fill byte. See the IAR

Part 1. Using the compiler
Linker and Library Tools Reference Guide for more information about the \(-H\) option and refer to the linker command file for an example of a suitable fill byte to use.

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 file includes 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 `0x2000-0xCFFF`.

\[-Z(CONST)MYSEGMENTA,MYSEGMENTB=2000-CFFF\]

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

\[-Z(CONST)MYSEGMENTA=2000-CFFF\]
\[-Z(CODE)MYCODE\]

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

\[-Z(CONST)MYSMALLSEGMENT=2000-20FF\]
\[-Z(CONST)MYLARGESEGMENT=2000-CFFF\]

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

**Using the \(-P\) command for packed placement**

The \(-P\) command differs from \(-Z\) in that it does not necessarily place the segments (or segment parts) sequentially. With \(-P\) it is possible to put segment parts into holes left by earlier placements.
The following example illustrates how the XLINK, -P option can be used for making efficient use of the memory area. This command will place the data segment MYDATA in DATA memory (that is, in RAM) in a fictitious memory range:

\[-P\text{(DATA)}\text{MYDATA}=0\text{-1FFF}\]

If your application has an additional RAM area in the memory range 10000\text{-11FFF}, you can simply add that to the original definition:

\[-P\text{(DATA)}\text{MYDATA}=0\text{-1FFF},10000\text{-11FFF}\]

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

**Note:** Copy initialization segments—BASENAME\_I and BASENAME\_ID—must be placed using -Z.

---

**Data segments**

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

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different data models available in the compiler. 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._2. 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:

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Segment group</th>
<th>Memory range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data16</td>
<td>DATA16</td>
<td>0x00000000-0x00007FFF and 0xFFFF8000-0xFFFFFFFF</td>
</tr>
<tr>
<td>Data24</td>
<td>DATA24</td>
<td>0x00000000-0x007FFFFF and 0xFF800000-0xFFFFFFFF</td>
</tr>
<tr>
<td>Data32</td>
<td>DATA32</td>
<td>0x00000000-0xFFFF0000</td>
</tr>
</tbody>
</table>

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 30.

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

<table>
<thead>
<tr>
<th>Categories of declared data</th>
<th>Suffix</th>
<th>Segment memory type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-initialized data</td>
<td>N</td>
<td>DATA</td>
</tr>
<tr>
<td>Zero-initialized data</td>
<td>Z</td>
<td>DATA</td>
</tr>
<tr>
<td>Non-zero initialized data</td>
<td>I</td>
<td>DATA</td>
</tr>
<tr>
<td>Initializers for the above</td>
<td>ID</td>
<td>CONST</td>
</tr>
<tr>
<td>Constants</td>
<td>C</td>
<td>CONST</td>
</tr>
<tr>
<td>Non-initialized absolute addressed data</td>
<td>AN</td>
<td></td>
</tr>
<tr>
<td>Constant absolute addressed data</td>
<td>AC</td>
<td></td>
</tr>
</tbody>
</table>

Table 11: Segment name suffixes

For a list of all supported segments, see Summary of segments, page 223.
Examples

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

```c
__data16 int j;
__data16 int i = 0;
__no_init __data16 int j;  // The data16 variables that are to be initialized to zero when the system starts are placed in the segment DATA16_Z.
__data16 int j = 4;  // 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.
```

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:

```c
/* First, the segments to be placed in ROM are defined. */
-Z(CONST) DATA16_C, DATA16_ID=FFFF8000-FFFFFFFF
-Z(CONST) DATA24_C, DATA24_ID=FFE00000-FFFFFFFF
-Z(CONST) DATA32_C, DATA32_ID=FFE00000-FFFFFFFF

/* Then, the RAM data segments are placed in memory. */
-Z(DATA) DATA16_N, DATA16_Z, DATA16_I=00000000-0001FFFF
-Z(DATA) DATA24_N, DATA24_Z, DATA24_I=00000000-0001FFFF
-Z(DATA) DATA32_N, DATA32_Z, DATA32_I=00000000-0001FFFF
```

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

**THE STACKS**

There are two stacks, the user mode stack and the supervisor mode stack. They are used by functions to store variables and other information that is used locally by functions, as described in the chapter Data storage. They are two continuous blocks of memory pointed to by the stack pointer registers USP and ISP.

The data segment used for holding the user mode stack is called USTACK and the data segment for the supervisor mode stack is called ISTACK. The system startup code initializes the stack pointers to the end of the stack segments.

The processor will be in supervisor mode on power on reset and when processing an interrupt. To enter user mode, special instruction sequences must be executed, as described in the chip manufacturer’s documentation.

The startup sequence in cstartup.s54 will remain in supervisor mode when calling the main function, so only the ISTACK segment will be used until the application enters user mode by its own means.

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

**Stack size allocation in the IDE**

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

### Stack size allocation from the command line

The size of the stack segments is defined in the linker command file.

The default linker command file sets up constants representing the sizes of the stacks, at the beginning of the file:

- `-D_USTACK_SIZE=size`
- `-D_ISTACK_SIZE=size`

**Note:** Normally, these lines are prefixed with the comment characters `//`. To make the directives take effect, remove the comment characters.

Specify appropriate stack sizes 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 segments are defined in the memory area available for the stacks:

- `-Z(DATA)USTACK+_USTACK_SIZE=00000000-0001FFFF`
- `-Z(DATA)ISTACK+_ISTACK_SIZE#00000000-0001FFFF`

**Note:**
- This range does not specify the size of the stack; it specifies the range of the available memory
- The `#` allocates the ISTACK 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 `_ISTACK_SIZE` bytes in size.

### Stack size considerations

The compiler uses the internal data stacks, USTACK and ISTACK, 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 HEAP

The heap contains dynamic data allocated by the C function malloc (or one of its relatives) or the C++ operator new.

If your application uses dynamic memory allocation, you should be familiar with:

- The linker segment 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=08000-08FFF

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
Placing code and data

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 RX microcontroller. 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 223.

**STARTUP CODE**

The segment `CSTART` contains code used during system startup and runtime initialization (`cstartup`), and system termination (`cexit`). The starting address of the system startup code is placed in the NMI reset vector entry, at address `0xFFFFFFFC`. The segments must also be placed into one continuous memory space, which means that the `-P` segment directive cannot be used.

In the default linker command file, this line will place the `CSTART` segment at the address `0xFFE00000`:

```
-Z(CODE)CSTART=FPE00000–FFFFFFFF
```

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

- P (CODE) CODE24=000000-FFFFFF
- P (CODE) CODE32=0-FFFFFFFF

Here, the -P linker directive is used for allowing 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 tables contain pointers to interrupt routines, including the reset routine. The tables are placed in the segments INTVEC and NMIVEC. For the RX microcontroller, you can place these segments anywhere in memory. The linker directives will look like this:

- Z(CONST) NMIVEC=FFFFFFD0-0001FFFF
- Z(CONST) INTVEC=FFE00000-FFFFFFFF

**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=FFE00000-FFFFFFFF

For additional information, see DIFUNCT, page 232.

**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*. 
Verifying the linked result of code and data placement
The DLIB runtime environment

This chapter describes the runtime environment in which an application executes. In particular, the chapter covers the DLIB runtime library and how you can 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 `rx\lib` and `rx\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.

**Note:** A separate floating-point library— iarfcmp.r54—is used for floating-point comparison. This library must be included if you link with libraries from another vendor than IAR Systems.

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.s54, 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, for example, locked registers
- 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 52.

LIBRARY CONFIGURATIONS

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

These DLIB library configurations are available:

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

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 52.

The prebuilt libraries are based on the default configurations, see Table 14, Prebuilt libraries, page 47. 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:

<table>
<thead>
<tr>
<th>Debugging support</th>
<th>Linker option in IDE</th>
<th>Linker command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic debugging</td>
<td></td>
<td></td>
<td>Debug support for C-SPY without any runtime support</td>
</tr>
<tr>
<td>Runtime debugging</td>
<td>Debug information</td>
<td><code>-P</code></td>
<td>The same as <code>-P</code>, but also includes debugger support for handling program abort, exit, and assertions.</td>
</tr>
<tr>
<td>I/O debugging</td>
<td>With I/O emulation</td>
<td><code>-rt</code></td>
<td>The same as <code>-r</code>, 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.</td>
</tr>
</tbody>
</table>

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 IAR C-SPY Debugger. For further information, see C-SPY runtime interface, page 66.

To set linker options for debug support in the IAR Embedded Workbench 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
- Size of the double floating-point type
- Byte order
- Library configuration—Normal or Full.
These prebuilt runtime libraries are available:

<table>
<thead>
<tr>
<th>Library</th>
<th>Code model</th>
<th>Size of double</th>
<th>Byte order</th>
<th>Library configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlrxffbf.r54</td>
<td>Far</td>
<td>32 bits</td>
<td>Big-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxffbn.r54</td>
<td>Far</td>
<td>32 bits</td>
<td>Big-endian</td>
<td>Normal</td>
</tr>
<tr>
<td>dlrxfflf.r54</td>
<td>Far</td>
<td>32 bits</td>
<td>Little-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxffln.r54</td>
<td>Far</td>
<td>32 bits</td>
<td>Little-endian</td>
<td>Normal</td>
</tr>
<tr>
<td>dlrxfdbf.r54</td>
<td>Far</td>
<td>64 bits</td>
<td>Big-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxfdbn.r54</td>
<td>Far</td>
<td>64 bits</td>
<td>Big-endian</td>
<td>Normal</td>
</tr>
<tr>
<td>dlrxfdlf.r54</td>
<td>Far</td>
<td>64 bits</td>
<td>Little-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxfdln.r54</td>
<td>Far</td>
<td>64 bits</td>
<td>Little-endian</td>
<td>Normal</td>
</tr>
<tr>
<td>dlrxhfbf.r54</td>
<td>Huge</td>
<td>32 bits</td>
<td>Big-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxhfbn.r54</td>
<td>Huge</td>
<td>32 bits</td>
<td>Big-endian</td>
<td>Normal</td>
</tr>
<tr>
<td>dlrxhflf.r54</td>
<td>Huge</td>
<td>32 bits</td>
<td>Little-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxhfln.r54</td>
<td>Huge</td>
<td>32 bits</td>
<td>Little-endian</td>
<td>Normal</td>
</tr>
<tr>
<td>dlrxhdbf.r54</td>
<td>Huge</td>
<td>64 bits</td>
<td>Big-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxhdbn.r54</td>
<td>Huge</td>
<td>64 bits</td>
<td>Big-endian</td>
<td>Normal</td>
</tr>
<tr>
<td>dlrxhdlf.r54</td>
<td>Huge</td>
<td>64 bits</td>
<td>Little-endian</td>
<td>Full</td>
</tr>
<tr>
<td>dlrxhdln.r54</td>
<td>Huge</td>
<td>64 bits</td>
<td>Little-endian</td>
<td>Normal</td>
</tr>
</tbody>
</table>

Table 14: Prebuilt libraries

The names of the libraries are constructed in this way:

\(<\text{type}><\text{cpu}><\text{code_model}><\text{size_double}><\text{byte_order}><\text{lib_config}>.r54\)

where

- \(<\text{type}>\) is \(dl\) for the IAR DLIB runtime environment
- \(<\text{cpu}>\) is \(rx\)
- \(<\text{code_model}>\) is either \(f\) for the Far code model or \(h\) for the Huge code model
- \(<\text{size_double}>\) is either \(f\) for 32 bits or \(d\) for 64 bits
- \(<\text{byte_order}>\) is either \(b\) for big-endian or \(l\) for little-endian
- \(<\text{lib_config}>\) is either \(n\) for the Normal configuration or \(f\) for the Full configuration.

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 these items to get the required runtime library:

- Specify which library object file to use on the XLINK command line, for instance: `dlrxhdlf.r54`
- Specify the include paths for the compiler and assembler:
  `-I rx\inc\`
- Specify the library configuration file for the compiler:
  `--dlib_config C:\...\dlrxhdlf.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 `rx\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:

<table>
<thead>
<tr>
<th>Items that can be customized</th>
<th>Described in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatters for <code>printf</code> and <code>scanf</code></td>
<td>Choosing formatters for <code>printf</code> and <code>scanf</code>, page 49</td>
</tr>
<tr>
<td>Startup and termination code</td>
<td>System startup and termination, page 54</td>
</tr>
<tr>
<td>Low-level input and output</td>
<td>Standard streams for input and output, page 57</td>
</tr>
<tr>
<td>File input and output</td>
<td>File input and output, page 61</td>
</tr>
<tr>
<td>Low-level environment functions</td>
<td>Environment interaction, page 64</td>
</tr>
<tr>
<td>Low-level signal functions</td>
<td>Signal and raise, page 65</td>
</tr>
<tr>
<td>Low-level time functions</td>
<td>Time, page 65</td>
</tr>
<tr>
<td>Size of heaps, stacks, and segments</td>
<td>Placing code and data, page 29</td>
</tr>
</tbody>
</table>

Table 15: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 51.
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 60.

**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:

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

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 60.

**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.
Choosing formatters for printf and scanf

**Specifying printf formatter from the command line**

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

- `-e_PrintfLarge=_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:

<table>
<thead>
<tr>
<th>Formatting capabilities</th>
<th>_ScanfFull</th>
<th>_ScanfLarge</th>
<th>_ScanfSmall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic specifiers c, d, i, o, p, s, u, u, X, X, and %</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Multibyte support</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>Floating-point specifiers a, and A</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Floating-point specifiers e, E, f, F, g, and G</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Conversion specifier n</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Scan set { and }</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment suppressing *</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>long long support</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 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 60.

**Specifying scanf formatter in the IDE**

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

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

- `e_ScanfLarge=_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 `rx\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:

   iccrx library_module

This creates a replacement object module file named library_module.r54.

Note: The code model, include paths, and the library configuration file must be the same for library_module as for the rest of your code.

4 Add library_module.r54 to the XLINK command line, either directly or by using an extended linker command file, for example:

   xlink library_module dlrxfdbf.r54

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.r54, 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 45, 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.

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

For information about the build process, see the IAR Embedded Workbench® IDE User Guide.

SETTING UP A LIBRARY PROJECT

The IDE provides library project templates which can be used for customizing the runtime environment configuration. These library templates have all available library configurations, see Table 12, Library configurations, page 45.

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 dlxlibaryname.h, which sets up that specific library with full library configuration. For more information, see Table 15, Customizable items, page 48.

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 dlxlibaryname.h and customize it by setting the values of the configuration symbols according to the application requirements.

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

USING A CUSTOMIZED LIBRARY

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

In the IDE you must do these steps:

1. Choose Project>Options and click the Library Configuration tab in the General Options category.
2. Choose Custom DLIB from the Library drop-down menu.
3. In the Library file text box, locate your library file.
4. In the Configuration file text box, locate your library configuration file.
System startup and termination

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

The code for handling startup and termination is located in the source files cstartup.s54, cexit.s54, and low_level_init.c located in the rx\src\lib directory.

For information about how to customize the system startup code, see Customizing system initialization, page 56.

SYSTEM STARTUP

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

For the hardware initialization, it looks like this:

- When the CPU is reset it will jump to the program entry label __program_start in the system startup code.
- The stack pointers, ISP, USP, and INTB, 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:

- 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 35.
- 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:

- 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 35.
- Static C++ objects are constructed.
- The main function is called, which starts the application.
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 66.

**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.s54` 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.s54 and low_level_init.c, located in the rx\src\lib directory.

**Note:** Normally, you do not need to customize the file cexit.s54.

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 52.

**Note:** Regardless of whether you modify the routine __low_level_init or the file cstartup.s54, you do not have to rebuild the library.

**__LOW_LEVEL_INIT**

A skeleton low-level initialization file is supplied with the product: low_level_init.c. 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.

The code calling __low_level_init at startup is only included if a module containing a __low_level_init definition is included when linking.

**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.S54**

As noted earlier, you should not modify the file cstartup.s54 if a customized version of __low_level_init is enough for your needs. However, if you do need to modify the file cstartup.s54, 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 51.

Note that you must make sure that the linker uses the same start label as used in your version of cstartup.s54. For information about how to change start label used by the linker, read about the linker option -s in the *IAR Linker and Library Tools Reference Guide*.

---

**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 rx\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 52. 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 66.

Example of using __write

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

```c
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 8;

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

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

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

    for (/* Empty */; bufSize > 0; --bufSize)
    {
        lcdIO = *buf;
        ++buf;
    }
```
The DLIB runtime environment

Part 1. Using the compiler

++nChars;
}
return nChars;
}

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

Example of using __read

The code in this example uses memory-mapped I/O to read from a keyboard:
#include <stddef.h>
__no_init volatile unsigned char kbIO @ 8;
size_t __read(int handle, unsigned char *buf, size_t bufSize)
{
    size_t nChars = 0;
    /* Check for stdin (only necessary if FILE descriptors are enabled) */
    if (handle != 0)
    {
        return -1;
    }
    for (/*Empty*/; bufSize > 0; --bufSize)
    {
        unsigned char c = kbIO;
        if (c == 0)
            break;
        *buf++ = c;
        ++nChars;
    }
    return nChars;
}

For information about the @ operator, see Controlling data and function placement in memory, page 105.
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 49.

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

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

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

<table>
<thead>
<tr>
<th>Config symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DLIB_PRINTF_MULTIBYTE</td>
<td>Multibyte characters</td>
</tr>
<tr>
<td>_DLIB_PRINTF_LONG_LONG</td>
<td>Long long (L qualifier)</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_FLOAT</td>
<td>Floating-point numbers</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_A</td>
<td>Hexadecimal floats</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_N</td>
<td>Output count (%n)</td>
</tr>
<tr>
<td>_DLIB_PRINTF_QUALIFIERS</td>
<td>Qualifiers h, j, l, L, v, t, and z</td>
</tr>
<tr>
<td>_DLIB_PRINTF_FLAGS</td>
<td>Flags -, +, #, and 0</td>
</tr>
<tr>
<td>_DLIB_PRINTF_WIDTH_AND_PRECISION</td>
<td>Width and precision</td>
</tr>
<tr>
<td>_DLIB_PRINTF_CHAR_BY_CHAR</td>
<td>Output char by char or buffered</td>
</tr>
</tbody>
</table>

Table 18: Descriptions of printf configuration symbols

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

<table>
<thead>
<tr>
<th>Config symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DLIB_SCANF_MULTIBYTE</td>
<td>Multibyte characters</td>
</tr>
<tr>
<td>_DLIB_SCANF_LONG_LONG</td>
<td>Long long (L qualifier)</td>
</tr>
<tr>
<td>_DLIB_SCANF_SPECIFIER_FLOAT</td>
<td>Floating-point numbers</td>
</tr>
<tr>
<td>_DLIB_SCANF_SPECIFIER_N</td>
<td>Output count (%n)</td>
</tr>
<tr>
<td>_DLIB_SCANF_QUALIFIERS</td>
<td>Qualifiers h, j, l, t, z, and L</td>
</tr>
<tr>
<td>_DLIB_SCANF.Scanset</td>
<td>Scanset (f)</td>
</tr>
<tr>
<td>_DLIB_SCANF_WIDTH</td>
<td>Width</td>
</tr>
<tr>
<td>_DLIB_SCANF_ASSIGNMENT_SUPPRESSING</td>
<td>Assignment suppressing (f)</td>
</tr>
</tbody>
</table>

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 52. 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 45. 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:

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

Table 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 46.
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.

**LOCALSEUPPORT IN PREBUILTLIBRARIES**

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

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

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

**CUSTOMIZING THE LOCALE SUPPORT**

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

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

**Locale configuration symbols**

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

```c
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C        /* C locale */
```
The DLIB runtime environment

#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 52.

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:

```c
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:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `rx\src\lib` directory. For information about overriding default library modules, see `Overriding library modules`, page 51.

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 52.
The DLIB runtime environment

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 46.

**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 `rx\src\lib` directory. For information about overriding default library modules, see Overriding library modules, page 51.

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 52.

**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 `rx\src\lib` directory. For information about overriding default library modules, see Overriding library modules, page 51.

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 52.

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 66.

**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 52. 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 `rx\src\lib` directory. For further information, see Building and using a customized library, page 52. 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**

All RX microcontroller devices are equipped with a single-precision hardware floating-point unit (FPU), that supports addition, subtraction, comparison, multiplication, division, and other instructions. The compiler will generate code that takes advantage of the FPU.

**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 46.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>C-SPY notifies that the application has called abort *</td>
</tr>
<tr>
<td>clock</td>
<td>Returns the clock on the host computer</td>
</tr>
<tr>
<td>__close</td>
<td>C-SPY notifies that the end of the application was reached *</td>
</tr>
<tr>
<td>__exit</td>
<td>C-SPY notifies that the end of the application was reached *</td>
</tr>
<tr>
<td>__open</td>
<td>Opens a file on the host computer</td>
</tr>
<tr>
<td>__read</td>
<td>std\input, std\output, and std\error will be directed to the Terminal I/O window; all other files will read the associated host file</td>
</tr>
<tr>
<td>remove</td>
<td>Writes a message to the Debug Log window and returns -1</td>
</tr>
<tr>
<td>rename</td>
<td>Writes a message to the Debug Log window and returns -1</td>
</tr>
</tbody>
</table>

*Table 21: Functions with special meanings when linked with debug info*
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 46. 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__ReportAssert</td>
<td>Handles failed asserts *</td>
</tr>
<tr>
<td>___seek</td>
<td>Seeks in the associated host file on the host computer</td>
</tr>
<tr>
<td>system</td>
<td>Writes a message to the Debug Log window and returns -1</td>
</tr>
<tr>
<td>time</td>
<td>Returns the time on the host computer</td>
</tr>
<tr>
<td>___write</td>
<td>stdin, stdout, and stderr will be directed to the Terminal I/O window, all other files will write to the associated host file</td>
</tr>
</tbody>
</table>

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.
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_buffereds__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.

<table>
<thead>
<tr>
<th>Object file</th>
<th>Color</th>
<th>Taste</th>
</tr>
</thead>
<tbody>
<tr>
<td>file1</td>
<td>blue</td>
<td>not defined</td>
</tr>
<tr>
<td>file2</td>
<td>red</td>
<td>not defined</td>
</tr>
<tr>
<td>file3</td>
<td>red</td>
<td>*</td>
</tr>
<tr>
<td>file4</td>
<td>red</td>
<td>spicy</td>
</tr>
<tr>
<td>file5</td>
<td>red</td>
<td>lean</td>
</tr>
</tbody>
</table>

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:

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

For detailed syntax information, see `rtmodel`, page 200.

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

```assembler
RTMODEL "color", 'red'
```

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

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

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

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

<table>
<thead>
<tr>
<th>Runtime model attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__rt_version</td>
<td>n</td>
<td>This runtime key is always present in all modules generated by the compiler. If a major change in the runtime characteristics occurs, the value of this key changes.</td>
</tr>
<tr>
<td>__core</td>
<td>RX600</td>
<td>The microcontroller core you are compiling for.</td>
</tr>
<tr>
<td>__double_size</td>
<td>32 or 64</td>
<td>The size, in bits, of the double floating-point type.</td>
</tr>
<tr>
<td>__endiian</td>
<td>1 or b</td>
<td>Corresponds to the byte order used in the project. 1 for little-endian or b for big-endian.</td>
</tr>
<tr>
<td>__calling_conv</td>
<td>ab10</td>
<td>Corresponds to the calling convention used in the project.</td>
</tr>
<tr>
<td>__lockRn</td>
<td>0 or 1</td>
<td>n is 8-13, for registers R8-R13. If the value is 0, the register is available for the compiler. If the value is 1, the register is locked for other uses.</td>
</tr>
</tbody>
</table>

Table 23: Predefined runtime model attributes

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 IAR Assembler Reference Guide for RX.

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, __lock_r13, is defined with a value counter. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not set this attribute.

Note that the compiler sets this attribute to free, unless the register is locked.
myCounter:
    add       #1.R13
    rts
    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 '__lock_r13' must be '1', 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 model1 and mode2. Declare this in each module that assumes that the UART is in a particular mode. This is how it could look like in one of the modules:

#pragma rtmodel="uart", "model"
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 RX microcontroller 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 RX 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 a function call and return instruction sequences, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all scratch registers are destroyed by an inline assembler instruction. In many cases, the overhead of the extra instructions can be removed by the optimizer.

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

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

- How should the assembler code be written so that it can be called from C?
- Where does the assembler code find its parameters, and how is the return value passed back to the caller?
- How should assembler code call functions written in C?
- How are global C variables accessed from code written in assembler language?
- Why does not the debugger display the call stack when assembler code is being debugged?
The first issue is discussed in the section *Calling assembler routines from C*, page 76. The following two are covered in the section *Calling convention*, page 79.

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 87.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 76, and *Calling assembler routines from C++*, page 78, 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.

```c
static int sFlag;

void foo(void)
{
    asm("mov.l #sFlag,R8");
    asm("mov.l #PIND,R9");
    while( !sFlag )
    {
        asm("mov.l [R9],[R8]*");
    }
}
```

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

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

Inline assembler sequences have no well-defined interface with the surrounding code generated from your C or C++ code. This makes the inline assembler code fragile, and 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:
  
  ```c
  extern int foo(void);
  
  or
  
  extern int foo(int i, int j);
  ```

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

**CREATING SKELETON CODE**

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

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

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

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

Note: In this example we use a low optimization level when compiling the code to show local and global variable access. If a higher level of optimization is used, the required references to local variables could be removed during the optimization. The actual function declaration is not changed by the optimization level.

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

Use these options to compile the skeleton code:

\texttt{iccrx 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 s54. Also remember to specify the code model and data model you are using, the byte order setting, the size of the double type, any locked registers, a low level of optimization, and -e for enabling language extensions.

The result is the assembler source output file \texttt{skeleton.s54}.

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. For more information, see *Call frame information*, page 87.

Calling assembler routines from C++

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

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

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

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

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.

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

class MyClass;

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

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

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

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

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

Calling convention

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

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

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

- Function declarations
- C and C++ linkage
- Preserved versus scratch registers
- Function entrance
Calling convention

- Function exit
- Return address handling.

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

**Note:** The calling convention complies with the RX ABI standard.

**FUNCTION DECLARATIONS**

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

```c
int 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:

```c
extern "C"
{
    int P(int);
}
```

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

```c
#ifdef __cplusplus
extern "C"
{
    int P(int);

    #endif
#endif
```

```c
#ifdef __cplusplus
#endif
```
PRESERVED VERSUS SCRATCH REGISTERS

The general RX CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

Any function is permitted to destroy the contents of a scratch register. If a function needs the register value after a call to another function, it must store it during the call, for example on the stack.

Any of the registers R1–R5 or R14–R15 can be used as a scratch register by the function.

Preserved registers

Preserved registers, on the other hand, are preserved across function calls. The called function can use the register for other purposes, but must save the value before using the register and restore it at the exit of the function.

The registers R6–R13 are preserved registers.

Special registers

For some registers, you must consider certain prerequisites:

● The stack pointer registers 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.

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. 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:

● Aggregate types (structures, unions and arrays) larger than 16 bytes, or with a lower alignment than 4

● 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 that does not fit into a register, the memory location where the structure will be stored is passed as the last function parameter.

Hidden parameters are passed in register R15.

Register parameters

The registers available for passing parameters are R1–R4.

The calling convention returns small aggregate types in registers R1–R4, if and only if they:

- are 16 bytes or smaller
- do not have an alignment less than 4.

Aggregate types that do not fit these two requirements will use a hidden parameter.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Passed in registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>8- to 32-bit values</td>
<td>R1–R4</td>
</tr>
<tr>
<td>64-bit values</td>
<td>R2R1, R3R2, R4R3</td>
</tr>
</tbody>
</table>

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 available, the parameter is passed on the stack. This process continues until there are no more parameter registers available or until all parameters have been passed.

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.

Objects on the stack should be aligned to 4 bytes at function entry, regardless of their
size.

When passed in registers, aggregate types follow the setting of the byte order option
--endian, but scalar types are always little-endian. On the stack, all parameters are
stored according to the byte order setting.

**FUNCTION EXIT**

A function can return a value to the function or program that called it, or it can have the
return type `void`.

The return value of a function, if any, can be scalar (such as integers and pointers),
floating-point, or a structure.

**Registers used for returning values**

The registers available for returning values are:

<table>
<thead>
<tr>
<th>Return values</th>
<th>Passed in registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>8- and 16-bit scalars</td>
<td>R1</td>
</tr>
<tr>
<td>32-bit values</td>
<td>R1</td>
</tr>
<tr>
<td>64-bit values</td>
<td>R2R1</td>
</tr>
<tr>
<td>Aggregate values</td>
<td>R1–R4</td>
</tr>
</tbody>
</table>

*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.

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 or RTSD instruction.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES
Interrupt functions save all used registers. Task functions save no registers at all, and monitor functions save the interrupt status.

An interrupt function returns by using the RTE instruction. Task functions and monitor functions return by using the RTS or RTSD instruction, depending on whether they need to deallocate a stack frame or not.

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 R1, and the return value is passed back to its caller in the register R1.

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

<table>
<thead>
<tr>
<th>name</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>rseg</td>
<td>CODE24:CODE</td>
</tr>
<tr>
<td>code</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>#1,R1</td>
</tr>
<tr>
<td></td>
<td>rts</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
Example 2
This example shows how structures are passed on the stack. Assume these declarations:

```c
struct MyStruct
{
    int mA;
    int mB;
};
int MyFunction(struct MyStruct x, int y);
```

The `struct` is passed in registers R1–R2, and the integer parameter `y` is passed in the register R3. The return value is passed back to its caller in the register R1. Compare with this example:

```c
struct MyStruct
{
    int mA;
    int mB;
    int mC;
    int mD;
    int mE;
};
int MyFunction(struct MyStruct x, int y);
```

Here, the calling function must reserve 20 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 R1. The return value is passed back to its caller in the register R1.

Example 3
The function below will return a structure of type `struct`.

```c
struct MyStruct
{
    int mA;
};
struct MyStruct MyFunction(int x);
```

In this case, the `struct` is small enough to fit in registers, so it is returned in R1.
Compare with this example:

```c
struct MyStruct
{
    int mA;
    int mB;
    int mC;
    int mD;
    int mE;
};

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

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 parameter. The pointer to the location where the return value should be stored is passed in R15.

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

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

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in R1, and the return value is returned in R1.

**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 RX 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 *IAR Assembler Reference Guide for RX*.
ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

This section presents the assembler instructions that can be used for calling and returning from functions on the RX microcontroller.

The default function calling instruction in the Far code model (and for functions explicitly declared __code24) is the BSR instruction:

```
bsr label
```

The location that the called function should return to (that is, the location immediately after this instruction) is stored on top of the stack. The destination label cannot be further away than 8 Mbytes.

The default function calling instruction in the Huge code model (and for functions explicitly declared __code32) is the JSR instruction, for example:

```
mov.l #label,Rn
jsr    Rn
```

Any call involving a __code32 function will first load the function address into a register and then call indirectly using the JSR instruction. This uses much more code space than calls between __code24 functions, and is also slower.

Call frame information

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

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

CFI DIRECTIVES

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

A full description about the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice.
When describing the call frame information, the following three components must be present:

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

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

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFA_SP</td>
<td>The call frame of the stack</td>
</tr>
<tr>
<td>R1–R15</td>
<td>Normal registers</td>
</tr>
<tr>
<td>SP</td>
<td>The stack pointer</td>
</tr>
<tr>
<td>PSW</td>
<td>The status register</td>
</tr>
<tr>
<td>?RET32</td>
<td>The return address</td>
</tr>
</tbody>
</table>

| Table 26: Call frame information resources defined in a names block |

**CREATING ASSEMBLER SOURCE WITH CFI SUPPORT**

The recommended way to create an assembler language routine that handles call frame information correctly is to start with an assembler language source file created by the compiler.

1. Start with suitable C source code, for example:

   ```c
   int F(int);  
   int cfiExample(int i)  
   {  
      return i + F(i);  
   }
   ```

2. Compile the C source code, and make sure to create a list file that contains call frame information—the CFI directives.

   - On the command line, use the option `-lA`.
   - In the IDE, choose *Project>Options>C/C++ Compiler> List* and make sure the suboption *Include compiler runtime information* is selected.
For the source code in this example, the list file looks like this, after it has been cleaned up for increased readability:

```
NAME Cfi

EXTERN F

PUBLIC _cfiExample
FUNCTION _cfiExample, 021203H

ARGFRAME CSTACK, 0, STACK
LOCFRAME CSTACK, 8, STACK

CFI Names cfiNames0
CFI StackFrame CFA SP DATA
CFI VirtualResource ?RET:32
CFI Resource SP:32, PSW:32
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 1
CFI DataAlign 1
CFI ReturnAddress ?RET CODE
CFI CFA SP+4
CFI ?RET Frame(CFA, -4)
CFI R1 Undefined
CFI R2 Undefined
CFI R3 Undefined
CFI R4 Undefined
CFI R5 Undefined
CFI R6 SameValue
CFI R7 SameValue
CFI R8 SameValue
CFI R9 SameValue
CFI R10 SameValue
CFI R11 SameValue
CFI R12 SameValue
CFI R13 SameValue
CFI R14 Undefined
CFI R15 Undefined
CFI PSW SameValue
CFI EndCommon cfiCommon0

RSEG CODE24:CODE:REORDER:NOROOT(0)
```
CFI Block cfiBlock0 Using cfiCommon0
CFI Function _cfiExample

CODE

_cfiExample:
  FUNCALL _cfiExample, _F
  LOCFRAME CSTACK, 8, STACK
  PUSH.L R6
  CFI R6 Frame(CFA, -8)
  CFI CFA SP+8
  MOV.L R1, R6
  BSR.A _F
  ADD R6, R1
  RTS 0x4, R6, R6
  CFI EndBlock cfiBlock0

END

Note: The header file cfi.m54 contains the macros XCFI_NAMES and XCFI_COMMON, which declare a typical names block and a typical common block. These two macros declare several resources, both concrete and virtual.
Using C++

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
Overview

IAR C/C++ Compiler

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 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 140.

To take advantage of Extended Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 140.
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 RX, 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 163.

**Example**

```c++
class MyClass
{
public:
   // Locate a static variable in __data16 memory at address 60
   static __data16 __no_init int mI @ 60;

   // Locate a static function in __code24 memory
   static __code24 void F();

   // Locate a function in __code24 memory
   __code24 void G();

   // Locate a virtual function in __code24 memory
   virtual __code24 void H();
};
```
// Locate a virtual function into SPECIAL
virtual void M() const volatile @ "SPECIAL";
};

Class memory
To compensate for this limitation, a class can be associated with a class memory type. The class memory type changes:

- the this pointer type in all member functions, constructors, and destructors into a pointer to class memory
- the default memory for static storage duration variables—that is, not auto variables—of the class type, into the specified class memory
- the pointer type used for pointing to objects of the class type, into a pointer to class memory.

Example
class __data16 C
{
public:
 void MyF(); // Has a this pointer of type C __data24 *
 void MyF() const; // Has a this pointer of type
 // C __data24 const *
 C(); // Has a this pointer pointing into data24 // memory
 C(C const &); // Takes a parameter of type C __data24 // memory
 // const & (also true of generated copy
 // constructor)
 int mI;
};

C Ca; // Resides in data24 memory instead of the
 // default memory
C __data16 Cb; // Resides in data16 memory, the ‘this’ // pointer still points into data24 memory

void MyH()
{
 C cd; // Resides on the stack
}

C *Cp1; // Creates a pointer to data24 memory
C __data16 *Cp2; // Creates a pointer to data16 memory
Whenever a class type associated with a class memory type, like \( C \), must be declared, the class memory type must be mentioned as well:

```cpp
class ___data24 C;
```

Also note that class types associated with different class memories are not compatible types.

A built-in operator returns the class memory type associated with a class, `__memory_of(class)`. For instance, `__memory_of(C)` returns `___data24`.

When inheriting, the rule is that it must be possible to convert implicitly a pointer to a subclass into a pointer to its base class. This means that a subclass can have a more restrictive class memory than its base class, but not a less restrictive class memory.

```cpp
class __data24 D : public C  // OK, same class memory
{ // OK, same class memory
  public:
    void MyG();
    int mJ;
  
  public:
    void MyG();  // Has a this pointer pointing into data16 memory
    MyF();     // Gets a this pointer into data24 memory
  int mJ;
  
  class __data16 E : public C
  { // OK, data16 memory is inside data24
    public:
      void MyG()  // Has a this pointer pointing into data16 memory
      {
        MyF();    // Gets a this pointer into data24 memory
      }
    int mJ;
  
  class __data24 F : public C
  { // OK, will be associated with same class memory as C
    public:
      void MyG();
      int mJ;
  
  A new expression on the class will allocate memory in the heap residing in the class memory. A delete expression will naturally deallocate the memory back to the same heap. To override the default `new` and `delete` operator for a class, declare

```cpp
void *operator new(size_t);
void operator delete(void *);
```

as member functions, just like in ordinary C++.

For more information about memory types, see Memory types, page 13.
FUNCTIONS

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

Example

```c
extern "C"
{
    typedef void (*FpC)(void);  // A C function typedef
}

typedef void (*FpCpp)(void);  // A C++ function typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
    MyF(F1);                      // Always works
    MyF(F2);                      // FpCpp is compatible with FpC
}
```

TEMPLATES

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

The standard template library

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

STL and the IAR C-SPY® Debugger

C-SPY has built-in display support for the STL containers. 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:

\[
\text{const\_cast\_t2>(t), static\_cast\_t2>(t), reinterpret\_cast\_t2>(t).}
\]

MUTABLE
The \texttt{mutable} attribute is supported in Extended EC++. A \texttt{mutable} symbol can be changed even though the whole class object is \texttt{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 \texttt{std} namespace.

THE STD NAMESPACE
The \texttt{std} namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the \texttt{std} namespace, simply define \texttt{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 \texttt{exit} or by returning from \texttt{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 \texttt{main} or when calling \texttt{exit} or \texttt{abort}.

To avoid interrupts, place a call to the intrinsic function \texttt{__disable\_interrupt} before the call to \texttt{__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:
  ```c++
  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:
  ```c++
  class A
  {
    const int mSize = 10; //Possible when using IAR language
                          //extensions
    int mArr[mSize];
  };
  ```
  According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:
  ```c++
  struct A
  {
    int A::F(); // Possible when using IAR language extensions
    int G();   // According to standard
  };
  ```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:
  ```c++
  extern "C" void F(); // Function with C linkage
  void (*PF)()         // PF points to a function with C++ linkage
                       = &F; // Implicit conversion of function pointer.
  ```
  According to the standard, the pointer must be explicitly converted.
● If the second or third operands in a construction that contains the ? operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to char * or wchar_t *, for example:

```cpp
bool X;
char *P1 = X ? "abc" : "def";       //Possible when using IAR
//language extensions
char const *P2 = X ? "abc" : "def"; //According to 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 canreference the non-static local variable. However, a warning is issued.

**Note:** If you use any of these constructions without first enabling language extensions, errors are issued.
C++ language extensions
Efficient coding for embedded applications

For embedded systems, the size of the generated code and data is very important, because using smaller external memory or on-chip memory can significantly decrease the cost and power consumption of a system.

The topics discussed are:

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Facilitating good code generation.

As a part of this, the chapter also demonstrates some of the more common mistakes and how to avoid them, and gives a catalog of good coding techniques.

Selecting data types

For efficient treatment of data, you should consider the data types used and the most efficient placement of the variables.

**USING EFFICIENT DATA TYPES**

The data types you use should be considered carefully, because this can have a large impact on code size and code speed.

- Use auto variables. Stack accesses are cheaper than global accesses, and many auto variables will end up in registers, making execution very fast.
- Use unsigned integer types where possible, unless your application really requires signed values. Many loop optimizations will work much better with unsigned loop variables.
- Try to avoid 64-bit data types, such as 64-bit double and long long.
Selecting data types

- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- Use signed or unsigned int for array indexing.
- Using floating-point types without using the built-in floating-point unit is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer to const data tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type float is more efficient in terms of code size and execution speed. However, the 64-bit format double supports higher precision and larger numbers.

In the compiler, the floating-point type float always uses the 32-bit format, and the format used by the double floating-point type depends on the setting of the --double compiler option.

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

By default, a floating-point constant in the source code is treated as being of the type double. This can cause innocent-looking expressions to be evaluated in double precision. In the example below a is converted from a float to a double, 1 is added and the result is converted back to a float:

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

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

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

For more information about floating-point types, see Floating-point types, page 158.
ALIGNMENT OF ELEMENTS IN A STRUCTURE

The RX microcontroller 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 155.

There are two ways to solve the problem:

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

For further details about the #pragma pack directive, see pack, page 198.

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 RX 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 139, for additional information.

Example

In this example, the members in the anonymous union can be accessed, in function f, without explicitly specifying the union name:

```c
struct S
{
    char mTag;
    union
```
The member names must be unique in the surrounding scope. Having an anonymous struct or union at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in this example:

```c
__no_init volatile union
{
  unsigned char IPORT;
  struct
  {
    unsigned char Way: 1;
    unsigned char Out: 1;
  };
} @ 8;
/* Here the variables are used*/

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

This declares an I/O register byte IPORT 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.

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_IPORT.
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 microcontroller 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 33, and Code segments, page 39, 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 30.
DATA PLACEMENT AT AN ABSOLUTE LOCATION

The @ operator, alternatively the #pragma location directive, can be used for placing global and static variables at absolute addresses. The variables must be declared using one of these combinations of keywords:

- __no_init
- __no_init and const (without initializers)
- const (with initializers).

To place a variable at an absolute address, the argument to the @ operator and the #pragma location directive should be a literal number, representing the actual address. The absolute location must fulfill the alignment requirement for the variable that should be located.

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

Examples

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

```c
__no_init volatile char alpha @ 0xFF2000; /* 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.

```c
#pragma location=0xFF2002
__no_init const int beta; /* OK */
const int gamma @ 0xFF2004 = 3; /* OK */
```

In the first case, the value is not initialized by the compiler; the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

These examples show incorrect usage:

```c
int delta @ 0xFF2008; /* Error, neither */
/* "__no_init" nor "const".*/
__no_init int epsilon @ 0xFF2007; /* Error, misaligned. */
```
C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```c
volatile const __no_init int x @ 0x100;        /* Bad in C++ */
```

the linker will report that more than one variable is located at address 0x100.

To avoid this problem and make the process the same in C and C++, you should declare these variables `extern`, for example:

```c
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 following method can be used for placing data or functions in named segments other than default:

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. The named segment can either be a predefined segment, or a user-defined segment. The variables must be declared either `__no_init` or `const`. If declared `const`, they can have initializers.

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.

```c
__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:

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

This example shows incorrect usage:

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

Examples of placing functions in named segments

```c
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:

```c
__data32 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 197, 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 144.

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 138.

OPTIMIZATION LEVELS

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

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (Best debug support)</td>
<td>Variables live through their entire scope</td>
</tr>
<tr>
<td></td>
<td>Dead code elimination</td>
</tr>
<tr>
<td></td>
<td>Redundant label elimination</td>
</tr>
<tr>
<td></td>
<td>Redundant branch elimination</td>
</tr>
<tr>
<td>Low</td>
<td>Same as above but variables only live for as long as they are needed, not necessarily through their entire scope</td>
</tr>
</tbody>
</table>

Table 27: Compiler optimization levels
 Controlling compiler optimizations

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see Fine-tuning enabled transformations, page 110.

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

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium</td>
<td>Same as above</td>
</tr>
<tr>
<td></td>
<td>Live-dead analysis and optimization</td>
</tr>
<tr>
<td></td>
<td>Code hoisting</td>
</tr>
<tr>
<td></td>
<td>Register content analysis and optimization</td>
</tr>
<tr>
<td></td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td></td>
<td>Static clustering</td>
</tr>
<tr>
<td>High (Balanced)</td>
<td>Same as above</td>
</tr>
<tr>
<td></td>
<td>Peephole optimization</td>
</tr>
<tr>
<td></td>
<td>Cross jumping</td>
</tr>
<tr>
<td></td>
<td>Cross call (when optimizing for size)</td>
</tr>
<tr>
<td></td>
<td>Loop unrolling</td>
</tr>
<tr>
<td></td>
<td>Function inlining</td>
</tr>
<tr>
<td></td>
<td>Code motion</td>
</tr>
<tr>
<td></td>
<td>Type-based alias analysis</td>
</tr>
</tbody>
</table>

Table 27: Compiler optimization levels (Continued)
line option --no_inline, alternatively its equivalent in the IDE Function inlining, or the #pragma optimize directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Static clustering
- 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 146.

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 148.

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.
Controlling compiler optimizations

**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 146.

**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 147.

**Example**

```c
short F(short *p1, long *p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the *short* pointed to
by *p1* cannot affect the *long* value that *p2* points to. Thus, it is known at compile time
that this function returns 0. However, in non-standard-conforming C or C++ code these
pointers could overlap each other by being part of the same union. If you use explicit
casts, you can also force pointers of different pointer types to point to the same memory location.

**Static clustering**

When static clustering is enabled, static and global variables that are defined within the same module are arranged so that variables that are accessed in the same function are stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses.

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

**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 145.

---

**Facilitating good code generation**

This section contains hints on how to allow the compiler to generate good code, for example:

- Using efficient addressing modes
- Helping the compiler optimize
- Generating more useful error messages.

**WRITING OPTIMIZATION-FRIENDLY CODE**

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the
optimizer can no longer assume that the local variable is unaffected over function calls.

- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.

- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining might enable further optimizations. The compiler often inlines small functions declared static. The use of the #pragma inline directive and the C++ keyword inline gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the --no_inline command line option; see --no_inline, page 146.

- 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 73.

**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.

**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 132. A higher alignment does not necessarily make the function faster, but the execution time will be more predictable.

**REGISTER LOCKING**

Register locking means that the compiler can be instructed never to touch some processor registers. This can be useful in several situations. For example:

- Some parts of a system could be written in assembler language to improve execution speed. These parts could be given dedicated processor registers.

- The register could be used by an operating system, or by other third-party software.
Registers are locked using the --lock compiler option. See --lock, page 143.

In general, if two modules are used together in the same application, they should have the same registers locked. The reason is that registers that can be locked could also be used as parameter registers when calling functions. In other words, the calling convention will depend on which registers that are locked.

To ensure that you only link modules with the same registers locked, you can use the __lockRn runtime model attribute; see Predefined runtime attributes, page 70.

FUNCTION PROTOTYPES

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

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

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

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

Prototyped style

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

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

int Test(char ch, int i) /* Definition */
{
    return i + ch;
}
```

Kernighan & Ritchie style

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

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

int Test(ch, i) /* Definition */
char ch;
int i;
{
    return i + ch;
}
```

### INTEGER TYPES AND BIT NEGATION

In some situations, the rules for integer types and their conversion lead to possibly confusing behavior. Things to look out for are assignments or conditionals (test expressions) involving types with different size, and logical operations, especially bit negation. Here, types also includes types of constants.

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 32-bit integer, and two’s complement is assumed:

```c
void F1(unsigned char c1)
{
    if (c1 == ~0x80)
        ;
}
```

Here, the test is always false. On the right hand side, 0x80 is 0x00000080, and -0x00000080 becomes 0xFFFFFFF7F. 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 163.

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 RX devices are included in the IAR product installation. The header files are named `iodevice.h` and define the processor-specific special function registers (SFRs).

**Note:** Each header file contains one section used by the compiler, and one section used by the assembler.

SFRs with bitfields are declared in the header file. This example is from `ior5f56108.h`:

```c
__no_init volatile union
{
  unsigned char SCI0_SMR;
  struct
  {
    unsigned char CKS  :  2;
    unsigned char      :  1;
    unsigned char STOP :  1;
    unsigned char PM   :  1;
    unsigned char PE   :  1;
    unsigned char CHR  :  1;
    unsigned char CM   :  1;
  } SCI0_SMR_bit;
} @ 0x00088240;
```

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

```c
void Test(void)
{
  /* whole register access */
  SCI0_SMR = 0x30;
  /* bit field accesses */
  SCI0_SMR_bit.STOP = 1;
  if(SCI0_SMR_bit.CKS)
```
You can also use the header files as templates when you create new header files for other RX devices. For details about the @ operator, see Located data, page 39.

**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 185. Note that to use this keyword, language extensions must be enabled; see -e, page 139. For information about the #pragma object_attribute, see page 197.
Part 2. Reference information

This part of the IAR C/C++ Compiler Reference Guide for RX 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:

```
icrx [options] [sourcefile] [options]
```

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

```
icrx 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 `iccrx` command, either before or after the source filename; see Invocation syntax, page 121.
Include file search procedure

This is a detailed description of the compiler’s `#include` file search procedure:

- If the name of the `#include` file is an absolute path, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:
  ```c
  #include <stdio.h>
  ```
  it searches these directories for the file to include:
  1. The directories specified with the `-I` option, in the order that they were specified, see `-I`, page 142.
  2. The directories specified using the `C_INCLUDE` environment variable, if any, see `Environment variables`, page 122.
- If the compiler encounters the name of an `#include` file in double quotes, for example:
  ```c
  #include "vars.h"
  ```
  it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.
If there are nested #include files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
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:

```
icrx ..\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:

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

Use angle brackets for standard header files, like stdio.h, and double quotes for files that are part of your application.

**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 r54.

- Optional list files
  
  Different types of list files can be specified using the compiler option -l. see -l, page 142. 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 124.

• 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 124.

• 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compilation successful, but there might have been warnings.</td>
</tr>
<tr>
<td>1</td>
<td>Warnings were produced and the option <code>--warnings_affect_exit_code</code> was used.</td>
</tr>
<tr>
<td>2</td>
<td>Errors occurred.</td>
</tr>
<tr>
<td>3</td>
<td>Fatal errors occurred, making the compiler abort.</td>
</tr>
<tr>
<td>4</td>
<td>Internal errors occurred, making the compiler abort.</td>
</tr>
</tbody>
</table>

*Table 29: 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:

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

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

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

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 153.

**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 148.

**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 130, 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 121.

**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:

```bash
--misracc2004=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:

```bash
-I..\src or -I ..\src\n
or
```

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:

```bash
--diagnostics_tables=MyDiagnostics.lst
```

or

```bash
--diagnostics_tables MyDiagnostics.lst
```

**Rules for options with both optional and mandatory parameters**

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

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

```bash
-lA MyList.lst
```

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

```bash
--preprocess=n PreprocOutput.lst
```

**Rules for specifying a filename or directory as parameters**

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

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

```bash
iccrx 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:

```
icrx 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:

```
icrx 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:

```
icrx 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`:

```
icrx 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:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--align_func</td>
<td>Specifies the alignment of function entry point</td>
</tr>
<tr>
<td>--char_is_signed</td>
<td>Treats char as signed</td>
</tr>
<tr>
<td>--code_model</td>
<td>Specifies the code model</td>
</tr>
<tr>
<td>--core</td>
<td>Specifies a CPU core</td>
</tr>
<tr>
<td>-D</td>
<td>Defines preprocessor symbols</td>
</tr>
<tr>
<td>--data_model</td>
<td>Specifies the data model</td>
</tr>
<tr>
<td>--debug</td>
<td>Generates debug information</td>
</tr>
<tr>
<td>--dependencies</td>
<td>Lists file dependencies</td>
</tr>
<tr>
<td>--diag_error</td>
<td>Treats these as errors</td>
</tr>
<tr>
<td>--diag_remark</td>
<td>Treats these as remarks</td>
</tr>
<tr>
<td>--diag_suppress</td>
<td>Suppresses these diagnostics</td>
</tr>
<tr>
<td>--diag_warning</td>
<td>Treats these as warnings</td>
</tr>
<tr>
<td>--diagnostics_tables</td>
<td>Lists all diagnostic messages</td>
</tr>
<tr>
<td>--discard_unused_publics</td>
<td>Discards unused public symbols</td>
</tr>
<tr>
<td>--dlib_config</td>
<td>Determines the library configuration file</td>
</tr>
<tr>
<td>--double</td>
<td>Forces the compiler to use 32-bit or 64-bit doubles</td>
</tr>
<tr>
<td>-e</td>
<td>Enables language extensions</td>
</tr>
<tr>
<td>--ec++</td>
<td>Enables Embedded C++ syntax</td>
</tr>
<tr>
<td>--eec++</td>
<td>Enables Extended Embedded C++ syntax</td>
</tr>
<tr>
<td>--enable_multibytes</td>
<td>Enables support for multibyte characters in source files</td>
</tr>
<tr>
<td>--endian</td>
<td>Specifies the byte order of the microcontroller</td>
</tr>
<tr>
<td>--error_limit</td>
<td>Specifies the allowed number of errors before compilation stops</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line</td>
</tr>
<tr>
<td>--header_context</td>
<td>Lists all referred source files and header files</td>
</tr>
<tr>
<td>-I</td>
<td>Specifies include file path</td>
</tr>
<tr>
<td>-l</td>
<td>Creates a list file</td>
</tr>
<tr>
<td>--library_module</td>
<td>Creates a library module</td>
</tr>
<tr>
<td>--lock</td>
<td>Locks registers</td>
</tr>
</tbody>
</table>

Table 30: Compiler options summary
### Compiler options

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--mfc</td>
<td>Enables multi file compilation</td>
</tr>
<tr>
<td>--module_name</td>
<td>Sets the object module name</td>
</tr>
<tr>
<td>--no_code_motion</td>
<td>Disables code motion optimization</td>
</tr>
<tr>
<td>--no_cross_call</td>
<td>Disables cross-call optimization</td>
</tr>
<tr>
<td>--no_cse</td>
<td>Disables common subexpression elimination</td>
</tr>
<tr>
<td>--no_inline</td>
<td>Disables function inlining</td>
</tr>
<tr>
<td>--no_path_in_file_macros</td>
<td>Removes the path from the return value of the symbols <strong>FILE</strong> and <strong>BASE_FILE</strong></td>
</tr>
<tr>
<td>--no_tbaa</td>
<td>Disables type-based alias analysis</td>
</tr>
<tr>
<td>--no_typedefs_in_diagnostics</td>
<td>Disables the use of typedef names in diagnostics</td>
</tr>
<tr>
<td>--no_unroll</td>
<td>Disables loop unrolling</td>
</tr>
<tr>
<td>--no_warnings</td>
<td>Disables all warnings</td>
</tr>
<tr>
<td>--no_wrap_diagnostics</td>
<td>Disables wrapping of diagnostic messages</td>
</tr>
<tr>
<td>-O</td>
<td>Sets the optimization level</td>
</tr>
<tr>
<td>-o</td>
<td>Sets the object filename</td>
</tr>
<tr>
<td>--omit_types</td>
<td>Excludes type information</td>
</tr>
<tr>
<td>--only_stdout</td>
<td>Uses standard output only</td>
</tr>
<tr>
<td>--output</td>
<td>Sets the object filename</td>
</tr>
<tr>
<td>--predef_macros</td>
<td>Lists the predefined symbols.</td>
</tr>
<tr>
<td>--preinclude</td>
<td>Includes an include file before reading the source file</td>
</tr>
<tr>
<td>--preprocess</td>
<td>Generates preprocessor output</td>
</tr>
<tr>
<td>--public_equ</td>
<td>Defines a global named assembler label</td>
</tr>
<tr>
<td>-r</td>
<td>Generates debug information</td>
</tr>
</tbody>
</table>

Table 30: 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}

Parameters

1 (default)  Sets the alignment of function entry points to 1 byte.
2  Sets the alignment of function entry points to 2 bytes.
4  Sets the alignment of function entry points to 4 bytes.

Description

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

See also

Aligning the function entry point, page 114.

Project>Options>C/C++ Compiler>Align functions
--char_is_signed

Syntax
--char_is_signed

Description
By default, the compiler interprets the char type as unsigned. Use this option to make the compiler interpret the char type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

Note: The runtime library is compiled without the --char_is_signed option. If you use this option, you might get type mismatch warnings from the linker, because the library uses unsigned char.

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

--code_model

Syntax
--code_model={far|huge}

Parameters
far (default) Functions can be placed in the high 8 Mbytes of memory
huge Functions can be placed anywhere

Description
Use this option to select the code model, that is, the default placement of functions. If you do not select a code model option, the compiler uses the default code model.

See also
Code models and memory attributes for function storage, page 21.

Project>Options>General Options>Target>Code model

--core

Syntax
--core=RX600

Parameters
RX600 Generates code for the RX600 microcontroller family

Description
This option is included for forward compatibility and has currently no effect.

This option is not available in the IDE.
Descriptions of options

-D

Syntax

-D symbol[=value]

Parameters

symbol The name of the preprocessor symbol
value The value of the preprocessor symbol

Description

Use this option to define a preprocessor symbol. If no value is specified, 1 is used. This option can be used one or more times on the command line.

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

-Dsymbol

is equivalent to:

#define symbol 1

To get the equivalence of:

#define FOO

specify the = sign but nothing after, for example:

-DFOO=

--data_model

Syntax

--data_model={near|far|huge}

Parameters

near Places variables and constant data in the lowest or highest 32 Kbytes of memory
far (default) Places variables and constant data in the lowest or highest 8 Mbytes of memory
huge Places variables and constant data anywhere in memory

Description

Use this option to select the data model, that is, the default placement of data objects. If you do not select a data model option, the compiler uses the default data model.

See also

Data models, page 12.
--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 128.

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.r54: c:\iar\product\include\stdio.h
foo.r54: 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:

```
%.r54 : %.c
 $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```
Descriptions of options

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

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

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

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

This option is not available in the IDE.

--diag_error

Syntax

   --diag_error=tag[,tag,...]

Parameters

   tag

   The number of a diagnostic message, for example the message
   number Pe117

Description

   Use this option to reclassify certain diagnostic messages as errors. An error indicates a
   violation of the C or C++ language rules, of such severity that object code will not be
   generated. The exit code will be non-zero. This option may be used more than once on
   the command line.

--diag_remark

Syntax

   --diag_remark=tag[,tag,...]

Parameters

   tag

   The number of a diagnostic message, for example the message
   number Pe177

Description

   Use this option to reclassify certain diagnostic messages as remarks. A remark is the
   least severe type of diagnostic message and indicates a source code construction that
   may cause strange behavior in the generated code. This option may be used more than
   once on the command line.
Note: By default, remarks are not displayed; use the --remarks option to display them.

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

--diag_suppress
Syntax
--diag_suppress=tag[,tag,...]
Parameters
tag
Description
Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.

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

--diag_warning
Syntax
--diag_warning=tag[,tag,...]
Parameters
tag
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.
Descriptions of options

**Description**
Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

This option is not available in the IDE.

---

**--discard_unused_publics**

**Syntax**
```
--discard_unused_publics
```

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

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

**Note:** Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output.

**See also**
--mfc, page 144 and Multi-file compilation units, page 109.

---

**--dlib_config**

**Syntax**
```
--dlib_config filename
```

**Parameters**
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 128.

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

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `rx\lib`. For examples and a list of prebuilt runtime libraries, see Using a prebuilt library, page 46.
If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see Building and using a customized library, page 52.

To set related options, choose:

Project>Options>General Options>Library Configuration

--double

Syntax

--double={32|64}

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 (default)</td>
<td>32-bit doubles are used</td>
</tr>
<tr>
<td>64</td>
<td>64-bit doubles are used</td>
</tr>
</tbody>
</table>

Description

Use this option to select the precision used by the compiler for representing the floating-point types double and long double. 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 158.

-e

Syntax

-e

Description

In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.

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

See also

The chapter Compiler extensions.

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* Extended Embedded C++, page 92.

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

--enable_multibyte

Syntax: `--enable_multibyte`

Description: By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer’s default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

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

--endian

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

Parameters:

- `b|big` Specifies big-endian as the default byte order for data
- `l|little` (default) Specifies little-endian as the default byte order for data
CRX-1

Part 2. Reference information

Compiler options

Description
Use this option to specify the byte order of the generated data. (Code is always little-endian.)

See also
Byte order, page 7.

Project>Options>General Options>Target>Byte order

--error_limit

Syntax
--error_limit=n

Parameters
n
The number of errors before the compiler stops the compilation. n must be a positive integer; 0 indicates no limit.

Description
Use the --error_limit option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.

This option is not available in the IDE.

-f

Syntax
-f filename

Parameters
For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 128.

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

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

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

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

--header_context

Syntax

--header_context

Description

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

This option is not available in the IDE.

-I

Syntax

-I path

Parameters

path The search path for #include files

Description

Use this option to specify the search paths for #include files. This option can be used more than once on the command line.

See also

Include file search procedure, page 122.

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

-l

Syntax

-l[a|A|b|B|c|D][N][H] {filename|directory}

Parameters

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.
Compiler options

CRX-1

Part 2. Reference information

---

**--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

**--lock**

**Syntax**

```
--lock={Ri|Rj,Rk|Rm-Rp}
```

**Parameters**

```
Ri|Rj,Rk|Rm-Rp    The register(s) to lock
```

**Description**

Use this option to lock one or several of the registers R8–R13 so that they cannot be used by the compiler but can be used for global register variables. To maintain module
consistency, make sure you lock the same registers in all modules. By default, no registers are locked.

Examples

```--lock=R10
--lock=R8,R12,R13
--lock=R10-R13
--lock=R8,R11-R13```

See also

*Register locking*, page 114.

---

### --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**

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

See also


---

### --module_name

**Syntax**

```--module_name=name```

**Parameters**

`name` An explicit object module name

**Description**

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.
This option is useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

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

---

### `--no_clustering`

**Syntax**

--no_clustering

**Description**

Use this option to disable static clustering optimizations. When static clustering is enabled, static and global variables are arranged so that variables that are accessed in the same function are stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses. These optimizations, which are performed at optimization levels Medium and High, normally reduce code size and execution time.

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

---

### `--no_code_motion`

**Syntax**

--no_code_motion

**Description**

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

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

---

### `--no_cross_call`

**Syntax**

--no_cross_call

**Description**

Use this option to disable the cross-call optimization. This optimization is performed at optimization level High, Size. Note that, although the option can drastically reduce the code size, this option increases the execution time.
Note: This option is not needed at optimization levels below High, or when optimizing Balanced or for Speed, because cross-call optimization is not enabled then.

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

--- **--no_cse**

**Syntax**

`--no_cse`

**Description**

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

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

--- **--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.
--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

This option is not available in the IDE.

--no_tbaa

Syntax
--no_tbaa

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

See also
Type-based alias analysis, page 112.

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

--no_typedefs_in_diagnostics

Syntax
--no_typedefs_in_diagnostics

Description
Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.

Example

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

will give an error message like this:

Error[Pe144]: a value of type 'char **' cannot be used to initialize an entity of type 'MyPtr'
If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type 'char *' cannot be used to initialize an entity of type 'int (*)(char const *)'
```

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

### --no_unroll

**Syntax**

`--no_unroll`

**Description**

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

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

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

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

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

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

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

### --no_warnings

**Syntax**

`--no_warnings`

**Description**

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

This option is not available in the IDE.
**--no_wrap_diagnostics**

**Syntax**

\[ --no_wrap_diagnostics \]

**Description**

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

This option is not available in the IDE.

**-O**

**Syntax**

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

**Parameters**

- **n** (None*) (Best debug support)
- **l** (default) (Low*)
- **m** (Medium)
- **h** (High, balanced)
- **hs** (High, favoring speed)
- **hz** (High, favoring size)

*The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.

**Description**

Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only \(-O\) is used without any parameter, the optimization level High balanced is used.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

**See also**

Controlling compiler optimizations, page 108.

Project>Options>C/C++ Compiler>Optimizations
-o, --output

Syntax

-o {filename|directory}
--output {filename|directory}

Parameters
For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 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 .r54. 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 Project>Options>C/C++ Compiler>Extra Options.

--only_stdout

Syntax

--only_stdout

Description
Use this option to make the compiler use the standard output stream (stdout) also for messages that are normally directed to the error output stream (stderr).

This option is not available in the IDE.
Compiler options

--output, -o

Syntax

--output {filename|directory}
-o {filename|directory}

Parameters

For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 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 .r54. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--predef_macros

Syntax

--predef_macros {filename|directory}

Parameters

For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 128.

Description

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

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

This option is not available in the IDE.

--preinclude

Syntax

--preinclude includefile

Parameters

For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 128.

Description

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

Project>Options>C/C++ Compiler>Preprocessor>Preinclude file
Descriptions of options

--preprocess
Syntax
```
--preprocess[=c][n][l] (filename|directory)
```
Parameters
- c: Preserve comments
- n: Preprocess only
- l: Generate #line directives

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

--public_equ
Syntax
```
--public_equ symbol[=value]
```
Parameters
- symbol: The name of the assembler symbol to be defined
- value: An optional value of the defined assembler symbol

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

Note: This option is not available in the IDE.

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

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

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

IAR C/C++ Compiler
Reference Guide
152
--remarks

**Syntax**

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**

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**

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 173. 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: --warnings_affect_exit_code

Description: By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code.

This option is not available in the IDE.

--warnings_are_errors

Syntax: --warnings_are_errors

Description: Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

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

See also diag_warning, page 296.

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

CRX-1
Data representation

This chapter describes the data types, pointers, and structure types supported by the compiler.

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

**Alignment**

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will have the same alignment as the structure member with the most strict alignment. To decrease the alignment requirements on the structure and its members, use `#pragma pack` or the `__packed` data type attribute.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

**ALIGNMENT ON THE RX MICROCONTROLLER**

The RX microcontroller can access memory using 8- to 32-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, ensuring that the RX microcontroller can read the data efficiently.
Basic data types

The compiler supports both all ISO/ANSI C basic data types and some additional types.

INTEGER TYPES

This table gives the size and range of each integer data type:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size</th>
<th>Range</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>8 bits</td>
<td>0 to 1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>8 bits</td>
<td>0 to 255</td>
<td>1</td>
</tr>
<tr>
<td>signed char</td>
<td>8 bits</td>
<td>-128 to 127</td>
<td>1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>0 to 255</td>
<td>1</td>
</tr>
<tr>
<td>signed short</td>
<td>16 bits</td>
<td>-32768 to 32767</td>
<td>2</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits</td>
<td>0 to 65535</td>
<td>2</td>
</tr>
<tr>
<td>signed int</td>
<td>32 bits</td>
<td>-2^31 to 2^31-1</td>
<td>4</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32 bits</td>
<td>0 to 2^31-1</td>
<td>4</td>
</tr>
<tr>
<td>signed long</td>
<td>32 bits</td>
<td>-2^31 to 2^31-1</td>
<td>4</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32 bits</td>
<td>0 to 2^32-1</td>
<td>4</td>
</tr>
<tr>
<td>signed long long</td>
<td>64 bits</td>
<td>-2^63 to 2^63-1</td>
<td>4</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64 bits</td>
<td>0 to 2^64-1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 31: Integer types

Signed variables are represented using the two’s complement form.

Bool

The bool data type is supported by default in the C++ language. If you have enabled language extensions, the bool type can also be used in C source code if you include the file stdbool.h. This will also enable the boolean values false and true.

The long long type

The long long data type is supported with one restriction: A long long variable cannot be used in a switch statement.

The enum type

The compiler will use the smallest type required to hold enum constants, preferring signed rather than unsigned.
When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example,

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

**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 RX, 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 RX, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size if --double=32</th>
<th>Size if --double=64</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>double</td>
<td>32 bits (default)</td>
<td>64 bits</td>
</tr>
<tr>
<td>long double</td>
<td>32 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

Table 32: Floating-point types

Note: The size of double and long double depends on the --double={32|64} option, see --double, page 139. The type long double uses the same precision as double.

Exception flags according to the IEEE 754 standard are not supported.

32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:

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

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

The value of the number is:

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

The range of the number is:

\[\pm1.18E-38 \text{ to } \pm3.39E+38\]

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:

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

The exponent is 11 bits, and the mantissa is 52 bits.
The value of the number is:

\[ (-1)^S \times 2^{(\text{Exponent}-1023)} \times 1.\text{Mantissa} \]

The range of the number is:

\[ \pm2.23\times10^{-308} \text{ to } \pm1.79\times10^{308} \]

The precision of the float operators (+, -, *, and /) is approximately 15 decimal digits.

**Representation of special floating-point numbers**

This list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- For the **float** type, 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.
- For the **double** type, Not a number (**NaN**) is represented by setting the exponent to **7FF** and at least one of the highest twenty bits in the mantissa to non-zero. The lower thirty-two bits of the mantissa are ignored. The value of the sign bit is also ignored.
- Subnormal numbers are supported for 64-bit floating-point numbers. For information about support for subnormal numbers for 32-bit floating-point numbers, see *Subnormal numbers*, page 159.

**Subnormal numbers**

Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is subnormal, even though the number is treated as if the exponent was 1. Unlike normal numbers, subnormal numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a subnormal number is:

\[ (-1)^S \times 2^{(1-\text{BIAS})} \times 0.\text{Mantissa} \]

where **BIAS** is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

By default, subnormal numbers are only supported for 64-bit floating-point numbers. However, the RX600 libraries can use the *unimplemented processing exception* of the CPU to support 32-bit floating-point subnormal numbers.
To enable the subnormal number exception handler, use the linker option
\texttt{-e} and use this linker command:

\texttt{-e \_unimpl\_processing\_handler=\_float\_placeholder}

Supporting subnormal numbers for 32-bit floating-point numbers this way requires a
large overhead, both in size and speed, compared to a normal FPU instruction which
requires very few CPU cycles. The subnormal number exception handler will use
approximately 900 bytes of code space, and about 50–200 cycles per exception,
depending on the operation and the operands. For that reason, if execution speed is
important, try to use floating-point algorithms that do not require subnormal number
capabilities for 32-bit floating-point numbers.

To remove subnormal number handling for 32-bit floating-point numbers, use this linker
command:

\texttt{-e \_floating\_point\_handler=\_float\_placeholder}

\section*{Pointer types}

The compiler has two basic types of pointers: function pointers and data pointers.

\subsection*{FUNCTION POINTERS}

The function pointer of the IAR C/C++ Compiler for RX is \texttt{\_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.

\subsection*{DATA POINTERS}

The data pointer of the IAR C/C++ Compiler for RX is \texttt{\_data32}. It is a 32-bit
\texttt{signed int} pointer that can address the entire memory.

\subsection*{CASTING}

Casts between pointers have these characteristics:

\begin{itemize}
  \item Casting a \texttt{value} of an integer type to a pointer of a smaller type is performed by
        truncation
  \item Casting a \texttt{value} of an integer type to a pointer of a larger type is performed by zero
        extension
  \item Casting a \texttt{pointer type} to a smaller integer type is performed by truncation
  \item Casting a \texttt{pointer type} to a larger integer type is performed by zero extension
  \item Casting a \texttt{data pointer} to a function pointer and vice versa is illegal
  \item Casting a \texttt{function pointer} to an integer type gives an undefined result
\end{itemize}
**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 RX, the size of `size_t` is 32 bits.

Note that for the Near data model, this is formally a violation of the standard; the size of `size_t` should actually be 16 bits.

**ptrdiff_t**

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the IAR C/C++ Compiler for RX, the size of `ptrdiff_t` is 32 bits.

For the Near data model, this is formally a violation of the standard; the size of `ptrdiff_t` should actually be 16 bits.

**intptr_t**

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for RX, the size of `intptr_t` is 32 bits.

**uintptr_t**

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned.

### Structure types

The members of a `struct` are stored sequentially in the order in which they are declared; the first member has the lowest memory address.

**ALIGNMENT**

The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. The size of a `struct` is also adjusted to allow arrays of aligned structure objects.

**GENERAL LAYOUT**

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

**Example**

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

} s;

This diagram shows the layout in memory:

![Figure 5: Structure layout](image)

The alignment of the structure is 2 bytes, and a pad byte must be inserted to give `short s` the correct alignment.

**PACKED STRUCTURE TYPES**

The `__packed` data type attribute or the `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work.

**Example**

This example declares a packed structure:

```c
#pragma pack(1)
struct S {
    char c;
    short s;
};

#pragma pack()
```

IAR C/C++ Compiler

162 Reference Guide
In this example, the structure \( S \) has this memory layout:

```
  0 1 2
  c s
```

*Figure 6: Packed structure layout*

This example declares a new non-packed structure, \( S_2 \), that contains the structure \( S \) declared in the previous example:

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

\( S_2 \) has this memory layout

```
  0 1 2 3 4 5 6 7
  c s pad l
```

*Figure 7: 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 \( S_2 \) will become 4.

For more information, see *Alignment of elements in a structure*, page 103.

---

**Type qualifiers**

According to the ISO/ANSI C standard, `volatile` and `const` are type qualifiers.

**DECLARING OBJECTS VOLATILE**

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.
**Definition of access to volatile objects**

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to volatile declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:
  ```c
  char volatile a;
  a = 5;  /* A write access */
  a += 6; /* First a read then a write access */
  ```
- An access to a bitfield is treated as an access to the underlaying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for RX are described below.

**Rules for accesses**

In the IAR C/C++ Compiler for RX, 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 memory types and all all properly aligned basic data types except 64-bit `double` and `long long`.

**DECLARING OBJECTS CONST**

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler’s possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` are always allocated in ROM.

In C++, objects that require runtime initialization cannot be placed in ROM.
Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.
Data types in C++
Compiler extensions

This chapter gives a brief overview of the compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- **C/C++ language extensions**
  For a summary of available language extensions, see *C language extensions*, page 168. 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 73. 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 216.

  **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 139, and `--strict_ansi`, page 154.

---

**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 105, and *location*, page 196.

- **Alignment**
  
  Each data type has its own alignment, for more details, see *Alignment*, page 155. If you want to change the alignment, the __packed data type attribute, the #pragma pack, and the #pragma data_alignment directive 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 103.

- **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 157.

- **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 __data32, the type of the __segment_begin operator is a pointer to __data32 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 __data32 *.

  ```
  #pragma segment="MYSEGMENT" __data32
  ...
  segment_start_address = __segment_begin('MYSECTION');
  ```
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 195.

- **Mixing declarations and statements**
  It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.

- **Declaration in for loops**
  It is possible to have a declaration in the initialization expression of a `for` loop, for example:
  ```c
  for (int i = 0; i < 10; ++i)
  {...}
  ```
  This feature is part of the C99 standard and C++.

- **The bool data type**
  To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)

- **C++ style comments**
  C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:
  ```c
  // The length of the bar, in centimeters.
  int length;
  ```
  This feature is copied from the C99 standard and C++.

**Inline assembler**

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict_ansi` is used. The `__asm` keyword is always available.

**Note:** Not all assembler directives or operators can be inserted using this keyword.
The syntax is:

```c
asm ("string");
```

The string can be a valid assembler instruction or a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```c
asm ("Label:    nop\n"      *   bra 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 73.

**Compound literals**

To create compound literals you can use this syntax:

```c
/* Create a pointer to an anonymous array */
int *p = (int []) {1,2,3};

/* Create a pointer to an anonymous structX */
structX *px = &(structX) {5,6,7};
```

Note:

- A compound literal can be modified unless it is declared `const`.
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

**Incomplete arrays at end of structs**

The last element of a `struct` can be an incomplete array. This is useful for allocating a chunk of memory that contains both the structure and a fixed number of elements of the array. The number of elements can vary between allocations.

This feature is part of the C99 standard.

Note: The array cannot be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

**Example**

```c
struct str
{
    char a;
    unsigned long b[];
};
```
struct str * GetAStr(int size)
{
    return malloc(sizeof(struct str) + 
                sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
    s->b[10] = 0;
}

The incomplete array will be aligned in the structure just like any other member of the structure. For more information about structure alignment, see Structure types, page 161.

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*2^2

Designated initializers in structures and arrays

Any initialization of either a structure (struct or union) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as .elementname and for an array [constant index expression]. Using designated initializers is not supported in C++.
Examples

This definition shows a struct and its initialization using designators:

```c
struct{
    int i;
    int j;
    int k;
    int l;
    short array[10];
} u = {
    .l = 6,          /* initialize l to 6 */
    .j = 6,          /* initialize j to 6 */
    8,               /* initialize k to 8 */
    .array[7] = 2,   /* initialize element 7 to 2 */
    .array[3] = 2,   /* initialize element 3 to 2 */
    5,               /* array[4] = 5 */
    .k = 4           /* reinitialize k to 4 */
};
```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```c
union {
    int i;
    float f;
} y = {.f = 5.0};
```

To set the size of an array by initializing the last element, do like this:

```c
char array[] = {
    [10] = 'a'
};
```

MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards issues and also some useful but minor syntax extensions:

- Arrays of incomplete types
  - An array can have an incomplete struct, union, or enum type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of enum types
  - The IAR Systems language extensions allow that you first declare the name of an enum and later resolve it by specifying the brace-enclosed list.
- Missing semicolon at end of \texttt{struct} or \texttt{union} specifier
  A warning is issued if the semicolon at the end of a \texttt{struct} or \texttt{union} specifier is missing.

- Null and \texttt{void}
  In operations on pointers, a pointer to \texttt{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 \textit{Casting}, page 160.

- 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, \texttt{short short} or \texttt{unsigned unsigned}), an error is issued.

- \texttt{long float} means \texttt{double}
  The type \texttt{long float} is accepted as a synonym for \texttt{double}.

- Repeated \texttt{typedef} declarations
  Redeclarations of \texttt{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, \texttt{unsigned char *} and \texttt{char *}. This includes pointers to integral types of the same size. A warning is issued.

- Non-top level \texttt{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, \texttt{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 RX, 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 RX, this expression is allowed:

  ```c
  struct str {
    int a;
    } x = 10;
  ```

• Declarations in other scopes
  External and static declarations in other scopes are visible. In the following example, the variable y can be used at the end of the function, even though it should only be visible in the body of the if statement. A warning is issued.

  ```c
  int test(int x) {
    if (x) {
      extern int y;
      y = 1;
    }
    return y;
  }
  ```
- Expanding function names into strings with the function as context

Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

`void func(char)`

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 139.
Extended keywords

This chapter describes the extended keywords that support specific features of the RX microcontroller 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 RX microcontroller. 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 181.

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 139 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 microcontroller.

- **Available function memory attributes**: `__code24` and `__code32`
- **Available data memory attributes**: `__data16`, `__data24`, and `__data32`

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:
  - `__fast_interrupt`, `__interrupt`, `__monitor`, `__task`
- **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 163].

**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` type 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:

```c
__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:

```c
#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:

```c
typedef char __data24 Byte;
Byte b;
```

```c
__data24 char b;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

### Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

```c
int __data24 * p; // The int object is located in data24 memory.
int * __data24 p;  // The pointer is located in data24 memory.
__data24 int * p;  // The pointer is located in data24 memory.
```

### Syntax for type attributes on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```c
__interrupt void my_handler(void);
```

or

```c
void (__interrupt my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

```c
#pragma type_attribute=__interrupt
void my_handler(void);
```
Syntax for type attributes on function pointers

To declare a function pointer, use this syntax:

```c
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:

```c
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`, `__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 105. For more information about `vector`, see *vector*, page 202.

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:

```c
__no_init int myarray[10];
```
The `#pragma object_attribute` directive can also be used. This declaration is equivalent to the previous one:

```c
#pragma object_attribute=__no_init
int myarray[10];
```

**Note:** Object attributes cannot be used in combination with the `typedef` keyword.

### Summary of extended keywords

This table summarizes the extended keywords:

<table>
<thead>
<tr>
<th>Extended keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__code24</td>
<td>Controls the storage of functions</td>
</tr>
<tr>
<td>__code32</td>
<td>Controls the storage of functions</td>
</tr>
<tr>
<td>__data16</td>
<td>Controls the storage of data objects</td>
</tr>
<tr>
<td>__data24</td>
<td>Controls the storage of data objects</td>
</tr>
<tr>
<td>__data32</td>
<td>Controls the storage of data objects</td>
</tr>
<tr>
<td>__fast_interrupt</td>
<td>Supports fast interrupt functions</td>
</tr>
<tr>
<td>__interrupt</td>
<td>Supports interrupt functions</td>
</tr>
<tr>
<td>__ intrinsic</td>
<td>Reserved for compiler internal use only</td>
</tr>
<tr>
<td>__monitor</td>
<td>Supports atomic execution of a function</td>
</tr>
<tr>
<td>__no_init</td>
<td>Supports non-volatile memory</td>
</tr>
<tr>
<td>__noreturn</td>
<td>Informs the compiler that the declared function will not return</td>
</tr>
<tr>
<td>__packed</td>
<td>Decreases data type alignment to 1</td>
</tr>
<tr>
<td>__root</td>
<td>Ensures that a function or variable is included in the object code even if unused</td>
</tr>
<tr>
<td>__task</td>
<td>Relaxes the rules for preserving registers</td>
</tr>
</tbody>
</table>

*Table 33: 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 *Type attributes*, page 177.
Descriptions of extended keywords

Description

The `__code24` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in code24 memory.

Storage information

- Address range: `0xFF800000–0xFFFFFFFF` (8 Mbytes)
- 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 177.

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 177.

Description

The `__data16` memory attribute overrides the default storage of variables and constants given by the selected data model, and places individual variables and constants in data16 memory.

Storage information

- Address range: `0–0x7FFF, 0xFFFF8000–0xFFFFFFFF` (64 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 Type attributes, page 177.

Description

The `__data24` memory attribute overrides the default storage of variables and constants given by the selected data model, and places individual variables and constants in data24 memory.

Storage information

- Address range: 0–0x7FFFFF, 0xFF800000–0xFFFFFFFF (16 Mbytes)
- Maximum object size: 8 Mbytes–1
- Pointer size: 4 bytes

Example

`__data24 int x;`

See also

Memory types, page 13.

---

**__data32**

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see Type attributes, page 177.

Description

The `__data32` memory attribute overrides the default storage of variables and constants given by the selected data model, and places individual variables and constants in data32 memory.

Storage information

- Address range: 0–0xFFFFFFFF (4 Gbytes)
- Maximum object size: 2 Gbytes–1
- Pointer size: 4 bytes.

Example

`__data32 int x;`

See also

Memory types, page 13.
__fast_interrupt

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 177.

Description
The __fast_interrupt keyword specifies a very fast interrupt function of the highest priority, using the FREIT return mechanism. 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

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 177.

Description
The __interrupt keyword specifies interrupt functions. To specify one or several interrupt vectors, use the #pragma vector directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table. An interrupt function must have a void return type and cannot have any parameters.

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

Example
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);

See also

__intrinsic

Description
The __intrinsic keyword is reserved for compiler internal use only.

__monitor

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 177.
Extended keywords

__monitor

**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 204, __enable_interrupt, page 204, __get_interrupt_state, page 205, and __set_interrupt_state, page 207.

__no_init

**Syntax**
Follows the generic syntax rules for object attributes, see Object attributes, page 180.

**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 180.

**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 177.

**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

```c
__packed struct X {char ch; int i;};          /* No pad bytes */
void foo (struct X * xp)              /* No need for __packed here */
{
    int * p1 = &xp->i; /* Error: int ** -> int __packed ** */
    int __packed * p2 = &xp->i; /* OK */
    char * p2 = &xp->ch;      /* OK, char not affected */
}
```

See also pack, page 198.

__root

Syntax Follows the generic syntax rules for object attributes, see Object attributes, page 180.

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.

__task

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 177.

Description This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the start function for a task in an RTOS.

By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared __task do not save all registers, and therefore require less stack space.
Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared `__task`.

Example

```c
__task void my_handler(void);
```
Pragma directives

This chapter describes the pragma directives of the compiler.

The #pragma directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the #pragma preprocessor directive or the _Pragma() preprocessor operator:

<table>
<thead>
<tr>
<th>Pragma directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitfields</td>
<td>Controls the order of bitfield members</td>
</tr>
<tr>
<td>constseg</td>
<td>Places constant variables in a named segment</td>
</tr>
<tr>
<td>data_alignment</td>
<td>Gives a variable a higher (more strict) alignment</td>
</tr>
<tr>
<td>dataset</td>
<td>Places variables in a named segment</td>
</tr>
<tr>
<td>diag_default</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_error</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_remark</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>diag_suppress</td>
<td>Suppresses diagnostic messages</td>
</tr>
<tr>
<td>diag_warning</td>
<td>Changes the severity level of diagnostic messages</td>
</tr>
<tr>
<td>include_alias</td>
<td>Specifies an alias for an include file</td>
</tr>
<tr>
<td>inline</td>
<td>Inlines a function</td>
</tr>
<tr>
<td>language</td>
<td>Controls the IAR Systems language extensions</td>
</tr>
<tr>
<td>location</td>
<td>Specifies the absolute address of a variable, or places groups of functions or variables in named segments</td>
</tr>
<tr>
<td>message</td>
<td>Prints a message</td>
</tr>
</tbody>
</table>

Table 34: Pragma directives summary
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.

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.
Pragma directives

bit. This setting remains active until you turn it off again with the #pragma bitfields=default directive.

See also  Bitfields, page 157.

cnstseg

Syntax  

#pragma constseg=[__memoryattribute ]{SEGMENT_NAME|default}

Parameters

__memoryattribute  An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.

SEGMENT_NAME  A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

default  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

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

Description  Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.
Note: Normally, the size of a variable is a multiple of its alignment. The data_alignment directive only affects the alignment of the variable’s start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

dataseg

Syntax

#pragma dataseg=[__memoryattribute ](SEGMENT_NAME|default)

Parameters

__memoryattribute An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.

SEGMENT_NAME 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 __no_init. The setting remains active until you turn it off again with the #pragma constseg=default directive.

Example

#pragma dataseg=__data24 MY_SEGMENT__no_init char myBuffer[1000];
#pragma dataseg=default

diag_default

Syntax

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

Parameters

tag The number of a diagnostic message, for example the message number Pe117.

Description

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

See also

Diagnostics, page 124.
**diag_error**

**Syntax**

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

**Parameters**

- `tag` The number of a diagnostic message, for example the message number Pe117.

**Description**

Use this pragma directive to change the severity level to *error* for the specified diagnostics.

**See also** *Diagnostics*, page 124.

---

**diag_remark**

**Syntax**

```c
#pragma diag_remark=tag[,tag,...]
```

**Parameters**

- `tag` The number of a diagnostic message, for example the message number Pe177.

**Description**

Use this pragma directive to change the severity level to *remark* for the specified diagnostic messages.

**See also** *Diagnostics*, page 124.

---

**diag_suppress**

**Syntax**

```c
#pragma diag_suppress=tag[,tag,...]
```

**Parameters**

- `tag` The number of a diagnostic message, for example the message number Pe117.

**Description**

Use this pragma directive to suppress the specified diagnostic messages.

**See also** *Diagnostics*, page 124.
**diag_warning**

Syntax

```c
#pragma diag_warning=tag1,tag2,...
```

Parameters

- `tag`: The number of a diagnostic message, for example the message number Pe826.

Description

Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages.

See also

Diagnostics, page 124.

**include_alias**

Syntax

```c
#pragma include_alias ("orig_header", "subst_header")
#pragma include_alias (<orig_header>, <subst_header>)
```

Parameters

- `orig_header`: The name of a header file for which you want to create an alias.
- `subst_header`: The alias for the original header file.

Description

Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file. This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```c
#pragma include_alias (<stdio.h>, <C:\MyHeaders\stdio.h>)
#pragma include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

See also

Include file search procedure, page 122.
inline

Syntax

#pragma inline[=forced]

Parameters

forced

Description

Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually occurs is subject to the compiler’s heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler’s heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

Note: Because specifying `#pragma inline=forced` disables the compiler’s heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.

language

Syntax

#pragma language={extended|default}

Parameters

extended

turns on the IAR Systems language extensions and turns off the
--strict_ansi command line option.
default

Uses the language settings specified by compiler options.

Description

Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.
Descriptions of pragma directives

**location**

**Syntax**

```
#pragma location={address|NAME}
```

**Parameters**

- **address**
  - The absolute address of the global or static variable for which you want an absolute location.

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

**Description**

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive.

**Example**

```
#pragma location=0xFF2000
__no_init volatile char PORT1; /* PORT1 is located at address 0xFF2000 */
```

```
#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 */
```

**See also**

*Controlling data and function placement in memory*, page 105.

**message**

**Syntax**

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

**Parameters**

- **message**
  - The message that you want to direct to `stdout`.

**Description**

Use this pragma directive to make the compiler print a message to `stdout` when the file is compiled.

**Example:**

```
#ifdef TESTING
#pragma message("Testing")
#endif
```
object_attribute

Syntax

#pragma object_attribute=object_attribute[, object_attribute,...]

Parameters

For a list of object attributes that can be used with this pragma directive, see Object attributes, page 180.

Description

Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive #pragma type_attribute that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.

Example

#pragma object_attribute=__no_init
char bar;

See also

General syntax rules for extended keywords, page 177.

optimize

Syntax

#pragma optimize=param[ param...]

Parameters

balanced|size|speed

Optimizes balanced between speed and size, optimizes for size, or optimizes for speed

none|low|medium|high

Specifies the level of optimization

no_code_motion

Turns off code motion

no_cse

Turns off common subexpression elimination

no_inline

Turns off function inlining

no_tbaa

Turns off type-based alias analysis

no_unroll

Turns off loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters speed, size, and balanced only have effect on the high optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.
Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```c
#pragma optimize=speed
int small_and_used_often()
{
    ... 
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
    ... 
}
```

**pack**

**Syntax**

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

**Parameters**

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

**Description**

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

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

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

See also

Structure types, page 161 and __packed, page 185.

__printf_args

Syntax

#pragma __printf_args

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

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

/* Function call */
printf("%d",x);  /* Compiler checks that x is a double */

required

Syntax

#pragma required=symbol

Parameters

symbol Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

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.

Example

const char copyright[] = "Copyright by me";
...
#pragma required=copyright
int main()
{
...}

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.
rtmodel

Syntax

```
#pragma rtmodel="key","value"
```

Parameters

*key*  
A text string that specifies the runtime model attribute.

*value*  
A text string that specifies the value of the runtime model attribute.

Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value *.* It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

**Note:** The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 68.

__scanf_args

Syntax

```
#pragma __scanf_args
```

Description

Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __scanf_args
int printf(char const *,...);
scanf("%d",x);  /* Compiler checks that x is a double */
```
Pragma directives

**segment**

**Syntax**

```
#pragma segment="NAME" [__memoryattribute] [align]
```

**Parameters**

- **NAME**
  The name of the segment
- **__memoryattribute**
  An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.
- **align**
  Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two.

**Description**

Use this pragma directive to define a segment name that can be used by the segment operators `__segment_begin` and `__segment_end`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

```
void __memoryattribute *
```

**Example**

```
#pragma segment="MYHUGE" __data32 4
```

**See also**

*Important language extensions*, page 168. For more information about segments and segment parts, see the chapter *Placing code and data.*

**type_attribute**

**Syntax**

```
#pragma type_attribute=[type_attribute[, type_attribute,...]]
```

**Parameters**

For a list of type attributes that can be used with this pragma directive, see *Type attributes*, page 177.

**Description**

Use this pragma directive to specify IAR-specific type attributes, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

**Example**

In this example, an int object with the memory attribute __data16 is defined:

```
#pragma type_attribute=__data16
int x;
```
This declaration, which uses extended keywords, is equivalent:

```c
__data16 int x;
```

See also See the chapter *Extended keywords* for more details.

### vector

**Syntax**

```c
#pragma vector=vector1[, vector2, vector3, ...]
```

**Parameters**

- `vector` 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!**

```c
#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:

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__break</td>
<td>Inserts a BRK instruction</td>
</tr>
<tr>
<td>__delay_cycles</td>
<td>Inserts code to delay execution</td>
</tr>
<tr>
<td>__disable_interrupt</td>
<td>Disables interrupts</td>
</tr>
<tr>
<td>__enable_interrupt</td>
<td>Enables interrupts</td>
</tr>
<tr>
<td>__exchange</td>
<td>Inserts an XCHG instruction</td>
</tr>
<tr>
<td>__get_FINTV_register</td>
<td>Returns the value of the FINTV register</td>
</tr>
<tr>
<td>__get_interrupt_level</td>
<td>Returns the interrupt level</td>
</tr>
<tr>
<td>__get_interrupt_state</td>
<td>Returns the interrupt state</td>
</tr>
<tr>
<td>__get_interrupt_table</td>
<td>Returns the value of the INTB register</td>
</tr>
<tr>
<td>__illegal_opcode</td>
<td>Inserts an illegal operation code</td>
</tr>
<tr>
<td>__no_operation</td>
<td>Inserts a NOP instruction</td>
</tr>
<tr>
<td>__RMPA_B</td>
<td>Inserts an RMPA.B instruction</td>
</tr>
<tr>
<td>__RMPA_L</td>
<td>Inserts an RMPA.L instruction</td>
</tr>
<tr>
<td>__RMPA_W</td>
<td>Inserts an RMPA.W instruction</td>
</tr>
<tr>
<td>__ROUND</td>
<td>Inserts a ROUND instruction</td>
</tr>
</tbody>
</table>

Table 35: Intrinsic functions summary
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 at least cycles number of execution cycles.

__disable_interrupt
Syntax
void __disable_interrupt(void);
Description
Disables interrupts by clearing the I bit in the processor status word register, PSW.

__enable_interrupt
Syntax
void __enable_interrupt(void);
Description
Enables interrupts by setting the I bit in the processor status word register, PSW.

Intrinsic function Description
__set_FINTV_register  Writes a specific value to the VCT register
__set_interrupt_level  Sets the interrupt level
__set_interrupt_state  Restores the interrupt state
__set_interrupt_table  Writes a specific value to the INTB register
__software_interrupt  Inserts an INT instruction
__wait_for_interrupt  Inserts a WAIT instruction

Table 35: Intrinsic functions summary (Continued)
Intrinsic functions

**__exchange**

**Syntax**

```c
unsigned long __exchange(unsigned long src, unsigned long * dst);
```

**Description**

Inserts an XCHG `src, dst` instruction.

**__get_FINTV_register**

**Syntax**

```c
unsigned long __get_FINTV_register(void);
```

**Description**

Returns the value of the FINTV register.

**__get_interrupt_level**

**Syntax**

```c
__ilevel_t __get_interrupt_level(void);
```

**Description**

Returns the current interrupt level. The return type `__ilevel_t` has this definition:

```c
typedef unsigned char __ilevel_t;
```

The return value of `__get_interrupt_level` can be used as an argument to the `__set_interrupt_level` intrinsic function.

**__get_interrupt_state**

**Syntax**

```c
__istate_t __get_interrupt_state(void);
```

**Description**

Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

**Example**

```c
__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 before the call of `__get_interrupt_state`.

---

**Part 2. Reference information** 205

CRX-1
__get_interrupt_table
Syntax  unsigned long __get_interrupt_table(void);
Description Returns the value of the INTB register.

__illegal_opcode
Syntax  void __illegal_opcode(void);
Description Inserts an illegal operation code.

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

__RMPA_B
Syntax  void __RMPA_B(signed char * v1, signed char * v2, unsigned long n, rmpa_t * acc);
Description Inserts an RMPA.B instruction. The RMPA instruction sequentially multiplies the two vectors v1 and v2 and adds each product to the accumulator acc. The length of the vectors is n. You can supply an initial value for the accumulator acc, either variable or a constant. The type rmpa_t is declared in the intrinsics.h file.

__RMPA_L
Syntax  void __RMPA_L(signed long * v1, signed long * v2, unsigned long n, rmpa_t * acc);
Description Inserts an RMPA.L instruction. The RMPA instruction sequentially multiplies the two vectors v1 and v2 and adds each product to the accumulator acc. The length of the vectors is n. You can supply an initial value for the accumulator acc, either variable or a constant. The type rmpa_t is declared in the intrinsics.h file.
Intrinsic functions

__RMPA_W

Syntax
void __RMPA_W(signed short * v1, signed short * v2, unsigned long n, rmpa_t * acc);

Description
Inserts an RMPA.W instruction. The RMPA instruction sequentially multiplies the two vectors v1 and v2 and adds each product to the accumulator acc. The length of the vectors is n. You can supply an initial value for the accumulator acc, either variable or a constant. The type rmpa_t is declared in the intrinsics.h file.

__ROUND

Syntax
int __ROUND(float);

Description
Inserts a ROUND instruction. See Casting a floating-point value to an integer, page 101.

__set_FINTV_register

Syntax
void __set_FINTV_register(unsigned long address);

Description
Writes a specific value to the FINTV register.

__set_interrupt_level

Syntax
void __get_interrupt_level(__ilevel_t);

Description
Sets the interrupt level. For information about the __ilevel_t type, see __get_interrupt_level, page 205.

__set_interrupt_state

Syntax
void __set_interrupt_state(__istate_t);

Description
Restores the interrupt state to a value previously returned by the __get_interrupt_state function.
For information about the __istate_t type, see __get_interrupt_state, page 205.
Descriptions of intrinsic functions

__set_interrupt_table
Syntax: void __set_interrupt_table(unsigned long address);
Description: Writes a specific value to the INTB register.

__software_interrupt
Syntax: void __software_interrupt(void);
Description: Inserts an INT instruction.

__wait_for_interrupt
Syntax: void __wait_for_interrupt(void);
Description: Inserts a WAIT 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 RX 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 210.

- **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 134.

- **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 212.

- **Preprocessor output**
  Use the option `--preprocess` to direct preprocessor output to a named file, see `--preprocess`, page 152.

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 240.
# Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

<table>
<thead>
<tr>
<th>Predefined symbol</th>
<th>Identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BASE_FILE</strong></td>
<td>A string that identifies the name of the base source file (that is, not the header file), being compiled. See also <strong>FILE</strong>, page 211, and --no_path_in_file_macros, page 147.</td>
</tr>
<tr>
<td><strong>BIG_ENDIAN</strong></td>
<td>An integer that identifies the setting of the option --endian. If --endian=b has been specified, the value of this symbol is defined to 1 (TRUE). If --endian=l has been specified, the value of this symbol is defined to 0 (FALSE).</td>
</tr>
<tr>
<td><strong>BUILD_NUMBER</strong></td>
<td>A unique integer that identifies the build number of the compiler currently in use.</td>
</tr>
<tr>
<td><strong>CODE_MODEL</strong></td>
<td>An integer that identifies the code model in use. The symbol reflects the --code_model option and can be defined to <strong>FAR</strong> or <strong>HUGE</strong>. These symbolic names can be used when testing the <strong>CODE_MODEL</strong> symbol.</td>
</tr>
<tr>
<td><strong>DATA_MODEL</strong></td>
<td>An integer that identifies the data model in use. The symbol reflects the --data_model option and can be defined to <strong>NEAR</strong>, <strong>FAR</strong>, or <strong>HUGE</strong>. These symbolic names can be used when testing the <strong>DATA_MODEL</strong> symbol.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>A string that identifies the date of compilation, which is returned in the form &quot;Mmm dd yyyy&quot;, for example &quot;Oct 30 2008&quot;.</td>
</tr>
<tr>
<td><strong>DOUBLE</strong></td>
<td>An integer that identifies the setting of the option --double. The symbol can be defined to 32 or 64.</td>
</tr>
<tr>
<td>__embedded_cplusplus</td>
<td>An integer which is defined to 1 when the compiler runs in any of the C++ modes, otherwise it is undefined. This symbol can be used with #ifdef to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*</td>
</tr>
</tbody>
</table>

*Note: The description of __embedded_cplusplus__ is marked with an asterisk to indicate a particular condition or limitation.
<table>
<thead>
<tr>
<th>Predefined symbol</th>
<th>Identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FILE</strong></td>
<td>A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also <strong>BASE_FILE</strong>, page 210, and --no_path_in_file_macros, page 147.*</td>
</tr>
<tr>
<td><strong>func</strong></td>
<td>A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 139. See also <strong>PRETTY_FUNCTION</strong>, page 211.</td>
</tr>
<tr>
<td><strong>FUNCTION</strong></td>
<td>A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 139. See also <strong>PRETTY_FUNCTION</strong>, page 211.</td>
</tr>
<tr>
<td><strong>IAR_SYSTEMS_ICC</strong></td>
<td>An integer that identifies the IAR compiler platform. The current value is 7. Note that the number could be higher in a future version of the product. This symbol can be tested with #ifdef to detect whether the code was compiled by a compiler from IAR Systems.</td>
</tr>
<tr>
<td><strong>ICCRX</strong></td>
<td>An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for RX, and otherwise to 0.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.*</td>
</tr>
<tr>
<td><strong>LITTLE_ENDIAN</strong></td>
<td>An integer that identifies the setting of the option --endian. If --endian=l has been specified, the value of this symbol is defined to 1 (TRUE). If --endian=b has been specified, the value of this symbol is defined to 0 (FALSE).</td>
</tr>
<tr>
<td><strong>PRETTY_FUNCTION</strong></td>
<td>A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example “void func(char)”. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 139. See also <strong>func</strong>, page 211.</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>An integer that is set to 1, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with #ifdef to detect whether the compiler in use adheres to ISO/ANSI C.*</td>
</tr>
</tbody>
</table>

Table 36: Predefined symbols (Continued)
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

Pragma("string")

where string follows the syntax of the corresponding pragma directive.

Description

This preprocessor operator is part of the C99 standard and can be used, for example, in defines and is equivalent to the #pragma directive.

Note: The -e option—enable language extensions—does not have to be specified.

Example

#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=none")
#else
  #define NOOPT
#endif

See also

See the chapterPragma directives.

#warning message

Syntax

#warning message

where message can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard #error directive is used.

__VA_ARGS__

Syntax

#define P(...) __VA_ARGS__
#define P(x,y,...) x + y + __VA_ARGS__

__VA_ARGS__ will contain all variadic arguments concatenated, including the separating commas.

Description

Variadic macros are the preprocessor macro equivalents of printf style functions. __VA_ARGS__ is part of the C99 standard.
Descriptions of miscellaneous preprocessor extensions

**Example**

```c
#if DEBUG
    #define DEBUG_TRACE(S,...) printf(S,__VA_ARGS__)
#else
    #define DEBUG_TRACE(S,...)
#endif

/* Place your own code here */
DEBUG_TRACE("The value is:%d\n",value);

will result in:

printf("The value is:%d\n",value);
```
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, mbstowc, mbsrtowc, wcrtomb, wcsrtomb
- The miscellaneous functions setlocale, rand, atexit, sterror, strtok
- Functions that use files in some way. This includes printf, scanf, getchar, and putchar. The functions sprintf and sscanf are not included.

Some functions also share the same storage for errno. These functions are not reentrant, since an errno value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. Among these functions are:

exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2, cosh, sinh, strtod, strtod, strtof, strto1

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 RX 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 220.

### C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Compiler extensions*.

The following table lists the C header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert.h</td>
<td>Enforcing assertions when functions execute</td>
</tr>
<tr>
<td>ctype.h</td>
<td>Classifying characters</td>
</tr>
<tr>
<td>errno.h</td>
<td>Testing error codes reported by library functions</td>
</tr>
<tr>
<td>float.h</td>
<td>Testing floating-point type properties</td>
</tr>
<tr>
<td>inttypes.h</td>
<td>Defining formatters for all types defined in stdint.h</td>
</tr>
<tr>
<td>iso646.h</td>
<td>Using Amendment 1—iso646.h standard header</td>
</tr>
<tr>
<td>limits.h</td>
<td>Testing integer type properties</td>
</tr>
<tr>
<td>locale.h</td>
<td>Adapting to different cultural conventions</td>
</tr>
<tr>
<td>math.h</td>
<td>Computing common mathematical functions</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>Executing non-local goto statements</td>
</tr>
<tr>
<td>signal.h</td>
<td>Controlling various exceptional conditions</td>
</tr>
<tr>
<td>stdarg.h</td>
<td>Accessing a varying number of arguments</td>
</tr>
<tr>
<td>stdbool.h</td>
<td>Adds support for the bool data type in C.</td>
</tr>
<tr>
<td>stddef.h</td>
<td>Defining several useful types and macros</td>
</tr>
<tr>
<td>stdint.h</td>
<td>Providing integer characteristics</td>
</tr>
<tr>
<td>stdio.h</td>
<td>Performing input and output</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>Performing a variety of operations</td>
</tr>
<tr>
<td>string.h</td>
<td>Manipulating several kinds of strings</td>
</tr>
<tr>
<td>time.h</td>
<td>Converting between various time and date formats</td>
</tr>
<tr>
<td>wchar.h</td>
<td>Support for wide characters</td>
</tr>
<tr>
<td>wctype.h</td>
<td>Classifying wide characters</td>
</tr>
</tbody>
</table>

*Table 37: Traditional standard C header files—DLIB*
C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex</td>
<td>Defining a class that supports complex arithmetic</td>
</tr>
<tr>
<td>exception</td>
<td>Defining several functions that control exception handling</td>
</tr>
<tr>
<td>fstream</td>
<td>Defining several I/O stream classes that manipulate external files</td>
</tr>
<tr>
<td>iomanip</td>
<td>Defining several I/O stream manipulators that take an argument</td>
</tr>
<tr>
<td>ios</td>
<td>Defining the class that serves as the base for many I/O streams classes</td>
</tr>
<tr>
<td>iosfwd</td>
<td>Defining several I/O stream classes before they are necessarily defined</td>
</tr>
<tr>
<td>iostream</td>
<td>Defining the I/O stream objects that manipulate the standard streams</td>
</tr>
<tr>
<td>istream</td>
<td>Defining the class that performs extractions</td>
</tr>
<tr>
<td>new</td>
<td>Defining several functions that allocate and free storage</td>
</tr>
<tr>
<td>ostream</td>
<td>Defining the class that performs insertions</td>
</tr>
<tr>
<td>sstream</td>
<td>Defining several I/O stream classes that manipulate string containers</td>
</tr>
<tr>
<td>stdexcept</td>
<td>Defining several classes useful for reporting exceptions</td>
</tr>
<tr>
<td>streambuf</td>
<td>Defining classes that buffer I/O stream operations</td>
</tr>
<tr>
<td>string</td>
<td>Defining a class that implements a string container</td>
</tr>
<tr>
<td>strstream</td>
<td>Defining several I/O stream classes that manipulate in-memory character sequences</td>
</tr>
</tbody>
</table>

Table 38: Embedded C++ header files

The following table lists additional C++ header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>fstream.h</td>
<td>Defining several I/O stream classes that manipulate external files</td>
</tr>
<tr>
<td>iomanip.h</td>
<td>Defining several I/O stream manipulators that take an argument</td>
</tr>
<tr>
<td>iostream.h</td>
<td>Defining the I/O stream objects that manipulate the standard streams</td>
</tr>
<tr>
<td>new.h</td>
<td>Defining several functions that allocate and free storage</td>
</tr>
</tbody>
</table>

Table 39: Additional Embedded C++ header files—DLIB
Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>Defines several common operations on sequences</td>
</tr>
<tr>
<td>deque</td>
<td>A deque sequence container</td>
</tr>
<tr>
<td>functional</td>
<td>Defines several function objects</td>
</tr>
<tr>
<td>hash_map</td>
<td>A map associative container, based on a hash algorithm</td>
</tr>
<tr>
<td>hash_set</td>
<td>A set associative container, based on a hash algorithm</td>
</tr>
<tr>
<td>iterator</td>
<td>Defines common iterators, and operations on iterators</td>
</tr>
<tr>
<td>list</td>
<td>A doubly-linked list sequence container</td>
</tr>
<tr>
<td>map</td>
<td>A map associative container</td>
</tr>
<tr>
<td>memory</td>
<td>Defines facilities for managing memory</td>
</tr>
<tr>
<td>numeric</td>
<td>Performs generalized numeric operations on sequences</td>
</tr>
<tr>
<td>queue</td>
<td>A queue sequence container</td>
</tr>
<tr>
<td>set</td>
<td>A set associative container</td>
</tr>
<tr>
<td>slist</td>
<td>A singly-linked list sequence container</td>
</tr>
<tr>
<td>stack</td>
<td>A stack sequence container</td>
</tr>
<tr>
<td>utility</td>
<td>Defines several utility components</td>
</tr>
<tr>
<td>vector</td>
<td>A vector sequence container</td>
</tr>
</tbody>
</table>

Table 40: Standard template library header files

Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, cassert and assert.h.

The following table shows the new header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassert</td>
<td>Enforcing assertions when functions execute</td>
</tr>
<tr>
<td>cctype</td>
<td>Classifying characters</td>
</tr>
<tr>
<td>cerrno</td>
<td>Testing error codes reported by library functions</td>
</tr>
<tr>
<td>cfloat</td>
<td>Testing floating-point type properties</td>
</tr>
<tr>
<td>cinttypes</td>
<td>Defining formatters for all types defined in stdint.h</td>
</tr>
</tbody>
</table>

Table 41: New standard C header files—DLIB
LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example memcpy, memset, and strcat.

ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality, partly taken from the C99 standard.

The following include files provide these features:

- <ctype.h>
- <inttypes.h>
- <math.h>
- <stdbool.h>
- <stdint.h>
- <stdio.h>
- <stdlib.h>
- <wchar.h>

Table 41: New standard C header files—DLIB (Continued)
• wctype.h

ctype.h
In ctype.h, the C99 function isblank is defined.

inttypes.h
This include file defines the formatters for all types defined in stdint.h to be used by the functions printf, scanf, and all their variants.

math.h
In math.h all functions exist in a float variant and a long double variant, suffixed by f and l respectively. For example, sinf and sinl.
The following C99 macro symbols are defined:
HUGE_VALF, HUGE_VALL, INFINITY, NAN, FP_INFINITE, FP_NAN, FP_NORMAL,
FP_SUBNORMAL, FP_ZERO, MATH_ERRNO, MATH_ERREXCEPT, math_errhandling.
The following C99 macro functions are defined:
fpclassify, signbit, isfinite, isnan, isnormal, isgreater, isless, islessequal, islessgreater, isunordered.
The following C99 type definitions are added:
float_t, double_t.

stdbool.h
This include file makes the bool type available if the Allow IAR extensions (-e) option is used.

stdint.h
This include file provides integer characteristics.

stdio.h
In stdio.h, the following C99 functions are defined:
vscanf, vfscanf, vsscanf, vsnprintf, snprintf
The functions printf, scanf, and all their variants have added functionality from the C99 standard. For reference information about these functions, see the library reference available from the Help menu.
The following functions providing I/O functionality for libraries built without FILE support are defined:

- `__write_array` Corresponds to `fwrite` on `stdout`.
- `__ungetchar` Corresponds to `ungetc` on `stdout`.
- `__gets` Corresponds to `fgets` on `stdin`.

**stdlib.h**

In `stdlib.h`, the following C99 functions are defined:

- `_Exit`, `llabs`, `lldiv`, `strtoll`, `strtoull`, `atoll`, `strtof`, `strtold`.

The function `strtod` has added functionality from the C99 standard. For reference information about this function, see the library reference available from the Help menu.

The `__qsortbbl` function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

**wchar.h**

In `wchar.h`, the following C99 functions are defined:

- `vfwscanf`, `vswscanf`, `vwscanf`, `wcstof`, `wcstolb`.

**wctype.h**

In `wctype.h`, the C99 function `iswblank` is defined.
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:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECKSUM</td>
<td>Holds the checksum generated by the linker.</td>
</tr>
<tr>
<td>CODE24</td>
<td>Holds __code24 program code.</td>
</tr>
<tr>
<td>CODE32</td>
<td>Holds __code32 program code.</td>
</tr>
<tr>
<td>CSTART</td>
<td>Holds the startup code.</td>
</tr>
<tr>
<td>DATA16_AC</td>
<td>Holds __data16 located constant data.</td>
</tr>
<tr>
<td>DATA16_AN</td>
<td>Holds __data16 located uninitialized data.</td>
</tr>
<tr>
<td>DATA16_C</td>
<td>Holds __data16 constant data.</td>
</tr>
<tr>
<td>DATA16_I</td>
<td>Holds __data16 static and global initialized variables.</td>
</tr>
<tr>
<td>DATA16_ID</td>
<td>Holds initial values for __data16 static and global variables in DATA16_I.</td>
</tr>
<tr>
<td>DATA16_N</td>
<td>Holds __no_init __data16 static and global variables.</td>
</tr>
<tr>
<td>DATA16_Z</td>
<td>Holds zero-initialized __data16 static and global variables.</td>
</tr>
<tr>
<td>DATA24_AC</td>
<td>Holds __data24 located constant data.</td>
</tr>
<tr>
<td>DATA24_AN</td>
<td>Holds __data24 located uninitialized data.</td>
</tr>
<tr>
<td>DATA24_C</td>
<td>Holds __data24 constant data.</td>
</tr>
<tr>
<td>DATA24_I</td>
<td>Holds __data24 static and global initialized variables.</td>
</tr>
<tr>
<td>DATA24_ID</td>
<td>Holds initial values for __data24 static and global variables in DATA24_I.</td>
</tr>
<tr>
<td>DATA24_N</td>
<td>Holds __no_init __data24 static and global variables.</td>
</tr>
<tr>
<td>DATA24_Z</td>
<td>Holds zero-initialized __data24 static and global variables.</td>
</tr>
</tbody>
</table>

Table 42: Segment summary
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 30.

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 31.

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 __code24 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 __code32 program code.
Segment memory type: CODE
Memory placement: This segment can be placed anywhere in memory.
Access type: Read-only

CSTART

Description: Holds the startup code.
This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.
Segment memory type: CODE
Memory placement: This segment must be placed in the highest 8 Mbytes of memory.
Descriptions of segments

**Access type**
Read-only

**DATA16_AC**

**Description**
Holds __data16 located constant data.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. Located means being placed at an absolute location using the @ operator or the #pragma location directive.

**DATA16_AN**

**Description**
Holds __no_init __data16 located data.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. Located means being placed at an absolute location using the @ operator or the #pragma location directive.

**DATA16_C**

**Description**
Holds __data16 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**
Holds __data16 static and global initialized variables initialized by copying from the segment DATA16_ID at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.

**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**
Holds initial values for __data16 static and global variables in the DATA16_I segment. These values are copied from DATA16_ID to DATA16_I at application startup. This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.

**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 __no_init __data16 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**
Holds zero-initialized __data16 static and global variables.
This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.

**Segment memory type**
DATA

**Memory placement**
This segment must be placed in the lowest 32 Kbytes of RAM memory.

**Access type**
Read/write
Descriptions of segments

**DATA24_AC**

Description: Holds __data24 located constant data.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. Located means being placed at an absolute location using the @ operator or the #pragma location directive.

**DATA24_AN**

Description: Holds __no_init __data24 located data.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. Located means being placed at an absolute location using the @ operator or the #pragma location directive.

**DATA24_C**

Description: Holds __data24 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: Holds __data24 static and global initialized variables initialized by copying from the segment DATA24_ID at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.

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**
Holds initial values for __data24 static and global variables in the DATA24_I segment. These values are copied from DATA24_ID to DATA24_I at application startup. This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.

**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 __no_init __data24 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**
Holds zero-initialized __data24 static and global variables. This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.

**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
Descriptions of segments

**DATA32_AC**

**Description**

Holds __data32 located constant data.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. Located means being placed at an absolute location using the @ operator or the #pragma location directive.

**DATA32_AN**

**Description**

Holds __no_init __data32 located data.

Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. Located means being placed at an absolute location using the @ operator or the #pragma location directive.

**DATA32_C**

**Description**

Holds __data32 constant data.

**Segment memory type**

`CONST`

**Memory placement**

This segment can be placed anywhere in ROM memory.

**Access type**

Read-only

**DATA32_I**

**Description**

Holds __data32 static and global initialized variables initialized by copying from the segment DATA32_ID at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used.

**Segment memory type**

`DATA`

**Memory placement**

This segment can be placed anywhere in RAM memory.

**Access type**

Read/write
**DATA32_ID**

| Description | Holds initial values for __data32 static and global variables in the DATA32_I segment. These values are copied from DATA32_ID to DATA32_I at application startup. This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z directive must be used. |
| 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 __no_init __data32 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 __data32 static and global variables. This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the -Z 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: CODE

Memory placement: In the Near data model, this segment must be placed in the first 64 Kbytes of memory. In other data models, this segment can be placed anywhere in memory.

Access type: Read-only

**HEAP**

Description: Holds the heap used for dynamically allocated data, in other words data allocated by malloc and free, and in C++, new and delete.

Segment memory type: DATA

Memory placement: In the Near data model, this segment must be placed in the first 64 Kbytes of memory. In other data models, this segment can be placed anywhere in memory.

Access type: Read/write

See also: The heap, page 38.

**INTVEC**

Description: Holds the interrupt vector table generated by the use of the __interrupt extended keyword in combination with the #pragma vector directive.

Segment memory type: CONST

Memory placement: This segment can be placed anywhere in ROM memory.

Access type: Read-only

**ISTACK**

Description: Holds the supervisor mode stack.

Segment memory type: DATA
<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
<th>Memory placement</th>
<th>Access type</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRX-1</td>
<td>This segment can be placed anywhere in RAM memory.</td>
<td>Read/write</td>
<td></td>
<td>The stacks, page 36.</td>
</tr>
<tr>
<td>NMIVEC</td>
<td>Holds the non-maskable interrupt vector table and the reset vector.</td>
<td>This segment must be placed in the memory range 0xFFFFFD0–0xFFFFFFFF.</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>SWITCH</td>
<td>Holds tables for switch statements. (Not all switch statements generate a table, but those who do will place the table in this segment.)</td>
<td>This segment can be placed anywhere in memory.</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>USTACK</td>
<td>Holds the user mode stack, referred to by the USP stack pointer.</td>
<td>This segment can be placed anywhere in RAM memory.</td>
<td>Read/write</td>
<td>The stacks, page 36.</td>
</tr>
</tbody>
</table>
Descriptions of segments
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, linenumeral level[tag]: message

where filename is the name of the source file in which the error was encountered, linenumeral 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 (5.1.2.2.2.1)
The function called at program startup is called main. No prototype was declared for main, and the only definition supported for main is:

```c
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see Customizing system initialization, page 56.

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 62.

**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 62.

**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 156, 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 158, 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 161, for information about size_t.

Conversion from/to pointers (6.3.4)
See Casting, page 160, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)
See pptrdiff_t, page 161, for information about the pptrdiff_t.

Registers

Honoring the register keyword (6.5.1)
User requests for register variables are not honored.

Structures, unions, enumerations, and bitfields

Improper access to a union (6.3.2.3)
If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

Padding and alignment of structure members (6.5.2.1)
See the section Basic data types, page 156, 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.
Descriptions of implementation-defined behavior

**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:

```c
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```c
file = fopen("mydirectory\myfile","rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter Pragma directives, the following directives are recognized and will have an indeterminate effect:

```c
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
```
Descriptions of implementation-defined behavior

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 mathematic 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 \texttt{fmod()} is zero, the function returns NaN; \texttt{errno} is set to \texttt{EDOM}.

\textbf{signal() (7.7.1.1)}
The signal part of the library is not supported.
\textbf{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 \textit{Signal and raise}, page 65.

Terminating newline character (7.9.2)
\texttt{stdout} stream functions recognize either \texttt{newline} or \texttt{end of file (EOF)} as the terminating character for a line.

Blank lines (7.9.2)
Space characters written to the \texttt{stdout} stream immediately before a newline character are preserved. There is no way to read the line through the \texttt{stdin} stream that was written through the \texttt{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 \textit{File input and output}, page 61.

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 \textit{File input and output}, page 61.

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 \textit{File input and output}, page 61.
%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 EPPOS in errno.

Message generated by perror() (7.9.10.4)
The generated message is:
usersuppliedprefix: errormessage

Allocating zero bytes of memory (7.10.3)
The calloc(), malloc(), and realloc() functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by realloc.

Behavior of abort() (7.10.4.1)
The abort() function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)
The argument passed to the exit function will be the return value returned by the main function to cstartup.

Environment (7.10.4.4)
The set of available environment names and the method for altering the environment list is described in Environment interaction, page 64.
**system() (7.10.4.5)**

How the command processor works depends on how you have implemented the `system` function. See Environment interaction, page 64.

**Message returned by strerror() (7.11.6.2)**

The messages returned by `strerror()` depending on the argument is:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EZERO</td>
<td>no error</td>
</tr>
<tr>
<td>EDOM</td>
<td>domain error</td>
</tr>
<tr>
<td>ERANGE</td>
<td>range error</td>
</tr>
<tr>
<td>EPPOS</td>
<td>file positioning error</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>multi-byte encoding error</td>
</tr>
<tr>
<td>`&lt;0</td>
<td></td>
</tr>
<tr>
<td>all others</td>
<td>error nnn</td>
</tr>
</tbody>
</table>

*Table 43: 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 65.

**clock() (7.12.2.1)**

From where the system clock starts counting depends on how you have implemented the `clock` function. See Time, page 65.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>Barr, Michael</td>
</tr>
<tr>
<td>implementation-defined behavior (DLIB)</td>
<td>247</td>
</tr>
<tr>
<td>system termination (DLIB)</td>
<td>baseaddr (pragma directive)</td>
</tr>
<tr>
<td>absolute location</td>
<td><strong>BASE_FILE</strong> (predefined symbol)</td>
</tr>
<tr>
<td>data, placing at (@)</td>
<td><strong>BIG_ENDIAN</strong> (predefined symbol)</td>
</tr>
<tr>
<td>language support for</td>
<td>binary streams (DLIB)</td>
</tr>
<tr>
<td>#pragma location</td>
<td>bit negation</td>
</tr>
<tr>
<td>addressing. See memory types, data models, and code models</td>
<td>assembler code</td>
</tr>
<tr>
<td>algorithm (STL header file)</td>
<td>calling from C</td>
</tr>
<tr>
<td>alignment</td>
<td>calling from C++</td>
</tr>
<tr>
<td>forcing stricter (#pragma data_alignment)</td>
<td>inserting inline</td>
</tr>
<tr>
<td>in structures (#pragma pack)</td>
<td>assembler directives, in inline assembler code</td>
</tr>
<tr>
<td>in structures, causing problems</td>
<td>assembler instructions</td>
</tr>
<tr>
<td>of an object (<strong>ALIGNOF</strong>)</td>
<td>inserting inline</td>
</tr>
<tr>
<td>of data types</td>
<td>used for calling functions</td>
</tr>
<tr>
<td>restrictions for inline assembler</td>
<td>assembler labels, making public (--public_equ)</td>
</tr>
<tr>
<td>alignment error, possible reason for</td>
<td>assembler language interface</td>
</tr>
<tr>
<td><strong>ALIGNOF</strong> (operator)</td>
<td>calling convention. See assembler code</td>
</tr>
<tr>
<td>--align_func (compiler option)</td>
<td>assembler list file, generating</td>
</tr>
<tr>
<td>anonymous structures</td>
<td>assembler output file</td>
</tr>
<tr>
<td>restrictions for inline assembler</td>
<td>assembler, inline</td>
</tr>
<tr>
<td>atomics</td>
<td>asserts</td>
</tr>
<tr>
<td>auto variables</td>
<td>implementation-defined behavior of (DLIB)</td>
</tr>
<tr>
<td>at function entrance</td>
<td>including in application</td>
</tr>
<tr>
<td>making accesses more efficient</td>
<td>assert.h (DLIB header file)</td>
</tr>
<tr>
<td>programming hints for efficient code</td>
<td>atoll, C99 extension</td>
</tr>
<tr>
<td>using in inline assembler code</td>
<td>atomic operations</td>
</tr>
<tr>
<td>attributes</td>
<td>__monitor</td>
</tr>
<tr>
<td>object</td>
<td>object</td>
</tr>
<tr>
<td>type</td>
<td>type</td>
</tr>
<tr>
<td>auto variables</td>
<td>17–18</td>
</tr>
<tr>
<td>at function entrance</td>
<td>81</td>
</tr>
<tr>
<td>making accesses more efficient</td>
<td>101</td>
</tr>
<tr>
<td>programming hints for efficient code</td>
<td>113</td>
</tr>
<tr>
<td>using in inline assembler code</td>
<td>76</td>
</tr>
</tbody>
</table>

**Index**
bitfields
   data representation of.........................157
   hints...........................................102
   implementation-defined behavior of ...........239
   non-standard types in........................169
   specifying order of members (#pragma bitfields).190
   bitfields (pragma directive)..................190
   bold style, in this guide........................xxv
   bool (data type)..................................156
   adding support for in DLIB ....................217, 220
   making available in C code .....................221
   __break (intrinsic function)....................204
   BRK (assembler instruction).....................204
   bubble sort function, defined in stdlib.h ......222
   building_runtime (pragma directive)..........241
   __BUILD_NUMBER__ (predefined symbol) .........210
   byte order ......................................xx
   choosing..........................................7
   identifying (__BIG_ENDIAN__) ...................210
   identifying (__LITTLE_ENDIAN__) ...............211
   cassert (DLIB header file) ......................219
   cast operators
      in Extended EC++ ...............................92
      missing from Embedded C++ ...................92
   casting, pointers and integers ................160
   cctype (DLIB header file) .......................219
   cerrno (DLIB header file) .......................219
   cexit (system termination code)
      in DLIB .......................................54
      placement in segment ..........................39
   CFI (assembler directive) .......................87
   cfi.m54 (CFI header example file) .............90
   cfloat (DLIB header file) .......................219
   char (data type) ..................................156
   changing default representation (--char_is_signed) 133
   signed and unsigned ................................157
   characters, implementation-defined behavior of ..236
   character-based I/O
      in DLIB .......................................58
      overriding in runtime library .................51
   –char_is_signed (compiler option) .............133
   CHECKSUM (segment) ..................................133
   climits (DLIB header file) ......................220
   classes ..........................................93
   clzits (DLIB header file) .......................220
   clocale (DLIB header file) ......................220
   clock (DLIB library function),
   implementation-defined behavior of ...........245
   clock.c ...........................................65
   __close (DLIB library function) ................61
   clustering (compiler transformation) ...........113
   disabling (--no_clustering) .................145
   cmath (DLIB header file) .......................220
   code
      execution of ....................................6
      interruption of execution ......................6
      verifying linked result .......................40
   code models .....................................21
   calling functions in ...........................86

C
C and C++ linkage ....................................80
C/C++ calling convention. See calling convention
C header files ......................................217
call frame information ...................................87
   in assembler list file ............................77
   in assembler list file (-IA) .................142
   call stack .......................................87
   callee-save registers, stored on stack .........18
   calling convention
      C++, requiring C linkage ......................78
      in compiler .....................................79
      __calling_conv (runtime model attribute) ......70
   calloc (library function) .......................19
   See also heap
      implementation-defined behavior of (DLIB) ........244
   can_instantiate (pragma directive) ..........244

IAR C/C++ Compiler
Reference Guide
248
<table>
<thead>
<tr>
<th>configuration</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifying (__CODE_MODEL__)</td>
<td>210</td>
</tr>
<tr>
<td>specifying on command line (--code_model)</td>
<td>133</td>
</tr>
<tr>
<td>code motion (compiler transformation)</td>
<td>112</td>
</tr>
<tr>
<td>disabling (--no_code_motion)</td>
<td>145</td>
</tr>
<tr>
<td>code pointers. See function pointers</td>
<td></td>
</tr>
<tr>
<td>code segments, used for placement</td>
<td>39</td>
</tr>
<tr>
<td>codeseg (pragma directive)</td>
<td>241</td>
</tr>
<tr>
<td>__CODE_MODEL__ (predefined symbol)</td>
<td>210</td>
</tr>
<tr>
<td>--code_model (compiler option)</td>
<td>133</td>
</tr>
<tr>
<td>_code24 (extended keyword)</td>
<td>182</td>
</tr>
<tr>
<td>CODE24 (segment)</td>
<td>225</td>
</tr>
<tr>
<td>_code32 (extended keyword)</td>
<td>182</td>
</tr>
<tr>
<td>_code32 (function pointer)</td>
<td>160</td>
</tr>
<tr>
<td>CODE32 (segment)</td>
<td>225</td>
</tr>
<tr>
<td>command line options</td>
<td></td>
</tr>
<tr>
<td>part of compiler invocation syntax</td>
<td>121</td>
</tr>
<tr>
<td>passing</td>
<td>121</td>
</tr>
<tr>
<td>See also compiler options</td>
<td></td>
</tr>
<tr>
<td>typographic convention</td>
<td>xxiv</td>
</tr>
<tr>
<td>commands</td>
<td></td>
</tr>
<tr>
<td>after preprocessor directives</td>
<td>175</td>
</tr>
<tr>
<td>C++ style, using in C code</td>
<td>170</td>
</tr>
<tr>
<td>common block (call frame information)</td>
<td>88</td>
</tr>
<tr>
<td>common subexpr elimination (compiler transformation)</td>
<td>111</td>
</tr>
<tr>
<td>disabling (--no_cse)</td>
<td>146</td>
</tr>
<tr>
<td>compilation date</td>
<td></td>
</tr>
<tr>
<td>exact time of (__TIME__)</td>
<td>212</td>
</tr>
<tr>
<td>identifying (__DATE__)</td>
<td>210</td>
</tr>
<tr>
<td>compiler</td>
<td></td>
</tr>
<tr>
<td>environment variables</td>
<td>122</td>
</tr>
<tr>
<td>invocation syntax</td>
<td>121</td>
</tr>
<tr>
<td>output from</td>
<td>123</td>
</tr>
<tr>
<td>compiler listing, generating (-l)</td>
<td>142</td>
</tr>
<tr>
<td>compiler object file, including debug information in (--debug, -r)</td>
<td>135</td>
</tr>
<tr>
<td>compiler optimization levels</td>
<td>109</td>
</tr>
<tr>
<td>compiler options</td>
<td>127</td>
</tr>
<tr>
<td>passing to compiler</td>
<td>121</td>
</tr>
<tr>
<td>reading from file (-f)</td>
<td>141</td>
</tr>
<tr>
<td>specifying parameters</td>
<td>129</td>
</tr>
<tr>
<td>summary</td>
<td>130</td>
</tr>
<tr>
<td>syntax</td>
<td>127</td>
</tr>
<tr>
<td>-l</td>
<td>77</td>
</tr>
<tr>
<td>--warnings_affect_exit_code</td>
<td>124</td>
</tr>
<tr>
<td>compiler platform, identifying</td>
<td>211</td>
</tr>
<tr>
<td>compiler subversion number</td>
<td>212</td>
</tr>
<tr>
<td>compiler transformations</td>
<td>108</td>
</tr>
<tr>
<td>compiler version number</td>
<td>212</td>
</tr>
<tr>
<td>compiling</td>
<td></td>
</tr>
<tr>
<td>from the command line</td>
<td>4</td>
</tr>
<tr>
<td>syntax</td>
<td>121</td>
</tr>
<tr>
<td>complex numbers, supported in Embedded C++</td>
<td>92</td>
</tr>
<tr>
<td>complex (library header file)</td>
<td>218</td>
</tr>
<tr>
<td>compound literals</td>
<td>171</td>
</tr>
<tr>
<td>computer style, typographic convention</td>
<td>xxiv</td>
</tr>
<tr>
<td>configuration</td>
<td></td>
</tr>
<tr>
<td>basic project settings</td>
<td>5</td>
</tr>
<tr>
<td>__low_level_init</td>
<td>57</td>
</tr>
<tr>
<td>configuration symbols, in library configuration files</td>
<td>53</td>
</tr>
<tr>
<td>consistency, module</td>
<td>68</td>
</tr>
<tr>
<td>const</td>
<td></td>
</tr>
<tr>
<td>declaring objects</td>
<td>164</td>
</tr>
<tr>
<td>non-top level</td>
<td>174</td>
</tr>
<tr>
<td>constants, placing in named segment</td>
<td>191</td>
</tr>
<tr>
<td>constseg (pragma directive)</td>
<td>191</td>
</tr>
<tr>
<td>const_cast (cast operator)</td>
<td>92</td>
</tr>
<tr>
<td>contents, of this guide</td>
<td>xxii</td>
</tr>
<tr>
<td>conventions, used in this guide</td>
<td>xxiv</td>
</tr>
<tr>
<td>copyright notice</td>
<td>ii</td>
</tr>
<tr>
<td>__core (runtime model attribute)</td>
<td>70</td>
</tr>
<tr>
<td>--core (compiler option)</td>
<td>133</td>
</tr>
<tr>
<td>core, specifying on command line</td>
<td>133</td>
</tr>
<tr>
<td>__cplusplus (predefined symbol)</td>
<td>210</td>
</tr>
<tr>
<td>cross call (compiler transformation)</td>
<td>113</td>
</tr>
<tr>
<td>csetjmp (DLIB header file)</td>
<td>220</td>
</tr>
<tr>
<td>csignal (DLIB header file)</td>
<td>220</td>
</tr>
<tr>
<td>csy_support (pragma directive)</td>
<td>241</td>
</tr>
</tbody>
</table>
CSTART (segment) ........................................... 39, 225
startup (system startup code) .................................. 39
customizing .................................................. 57
overriding in runtime library .................................. 51
startup.s54 .................................................. 54
cstdlib (DLIB header file) .................................. 220
cstdio (DLIB header file) .................................. 220
cstring (DLIB header file) .................................. 220
cdate (DLIB header file) .................................. 220
type.h (library header file) .................................. 217
added C functionality ........................................ 221
cwchar.h (library header file) .............................. 220
C_INCLUDE (environment variable) ........................... 122
C-SPY
interface to system termination ............................... 56
low-level interface .......................................... 66
STL container support ........................................ 96
C++

See also Embedded C++ and Extended Embedded C++
absolute location ........................................... 107
calling convention .......................................... 78
dynamic initialization in .................................... 40
features excluded from EC++ ................................ 91
header files .................................................. 218
language extensions .......................................... 98
special function types ....................................... 27
static member variables .................................... 107
support for .................................................. 3
C++ names, in assembler code ................................ 79
C++ objects, placing in memory type ......................... 17
C++ terminology ........................................... xxiv
C++-style comments .......................................... 170
C99 standard, added functionality from ................... 220

D
-D (compiler option) ......................................... 134
--data_model (compiler option) ............................. 134
data
alignment of ................................................ 155
different ways of storing .................................... 11
located, declaring extern .................................. 107
placing ....................................................... 105, 192, 223
  at absolute location ...................................... 106
representation of ........................................... 155
storage ....................................................... 11
verifying linked result ...................................... 40
data block (call frame information) ......................... 88
data memory attributes, using .............................. 14
data models .................................................. 12
configuration ............................................... 6
identifying (__DATA_MODEL__) ............................. 210
data pointers ............................................... 160
data segments .............................................. 33
data types .................................................. 156
  floating point ........................................... 158
  in C++ ................................................... 165
  integers .................................................. 156
dataseg (pragma directive) ................................ 192
data_alignment (pragma directive) ......................... 191
__DATA_MODEL__ (predefined symbol) ..................... 210
__data16 (extended keyword) ................................ 182
data16 (memory type) ..................................... 13
DATA16_AC (segment) .................................... 226
DATA16_AN (segment) .................................... 226
DATA16_C (segment) .................................... 226
DATA16_I (segment) .................................... 226
DATA16_ID (segment) .................................... 227
DATA16_N (segment) .................................... 227
DATA16_Z (segment) .................................... 227
__data24 (extended keyword) ................................ 183
data24 (memory type) ..................................... 14
DATA24_AC (segment) .................................... 228
Index

DATA24_AN (segment) ................................. 228
DATA24_C (segment) ................................. 228
DATA24_I (segment) ................................. 229
DATA24_ID (segment) ................................. 229
DATA24_N (segment) ................................. 229
DATA24_Z (segment) ................................. 229
__data32 (extended keyword) ..................... 183
__data32 (data pointer) ......................... 160
data32 (memory type) ............................. 14
DATA32_AC (segment) .............................. 230
DATA32_AN (segment) .............................. 230
DATA32_C (segment) .............................. 230
DATA32_I (segment) .............................. 230
DATA32_ID (segment) .............................. 231
DATA32_N (segment) .............................. 231
DATA32_Z (segment) .............................. 231
__DATE__ (predefined symbol) ................. 210
date (library function), configuring support for .......................... 65
DC32 (assembler directive) ..................... 76
--debug (compiler option) .................... 135
declarations, including in object file ........ 135, 152
declarations
empty ............................................. 175
in for loops ..................................... 170
Kernighan & Ritchie ............................... 115
of functions ..................................... 80
declarations and statements, mixing ........ 170
declarators, implementation-defined behavior .... 240
define_type_info (pragma directive) ......... 241
delay code, inserting ......................... 204
__delay_cycles (intrinsic function) ......... 204
delete (keyword) .................................. 19
denormalized numbers. See subnormal numbers
--dependencies (compiler option) .......... 135
dequeue (STL header file) ................. 219
destructors and interrupts, using .......... 97
diagnostic messages ......................... 124
classifying as compilation errors ........ 136
classifying as compilation remarks ........ 136
classifying as compiler warnings .......... 137
disabling compiler warnings ............... 148
disabling wrapping of in compiler .......... 149
enabling compiler remarks ................ 153
listing all used by compiler ............... 137
suppressing in compiler ...................... 137
--diagnostics_tables (compiler option) .... 137
diag__default (pragma directive) ........ 192
--diag_error (compiler option) .......... 136
diag__error (pragma directive) ........... 193
--diag_remark (compiler option) .......... 136
diag__remark (pragma directive) ......... 193
--diag_suppress (compiler option) ....... 137
diag__suppress (pragma directive) ....... 193
--diag_warning (compiler option) ........ 137
diag__warning (pragma directive) ....... 194
DIFUNCT (segment) ......................... 40, 232
directives
function for static overlay ............... 86
pragma ........................................ 9, 189
directory, specifying as parameter .......... 128
__disable_interrupt (intrinsic function) .... 204
--discard_unused_publics (compiler option) .... 138
disclaimer ........................................ ii
DLIB ........................................... 7, 216
building customized library ............... 45
configurations .................................. 45
configuring ..................................... 44, 138
debug support ................................... 46
reference information. See the online help system .... 215
runtime environment ......................... 43
--dlib_config (compiler option) .......... 138
Dlib_defaults.h (library configuration file) .... 53
dlrXlibname.h .................................. 53
document conventions ........................ xxiv
documentation, library ....................... 215
domain errors, implementation-defined behavior .... 242
__DOUBLE__ (predefined symbol) .......... 210
--double (compiler option) ............... 139

251
double (data type) ........................................... 158
avoiding ............................................. 101
configuring size of floating-point type ............... 6
__double_size (runtime model attribute) ............. 70
double_t, C99 extension ................................. 221
do_not_instantiate (pragma directive) ................... 241
dynamic initialization .................................. 54
in C++ .................................................. 40
dynamic memory ......................................... 19

E

-e (compiler option) ..................................... 139
early_initialization (pragma directive) ................... 241
--ec++ (compiler option) ................................ 140
EC++ header files ........................................ 218
edition, of this guide .................................. ii
--ec++ (compiler option) ................................ 140
Embedded C++ ........................................... 91
differences from C++ ..................................... 91
enabling ................................................ 140
function linkage ......................................... 80
language extensions ..................................... 91
overview ............................................... 91
Embedded C++ Technical Committee .................. xxiv
embedded systems, IAR special support for .......... 9
__embedded_cplusplus (predefined symbol) .......... 210
__enable_interrupt (intrinsic function) ................. 204
__enable_multibytes (compiler option) ................. 140
__endian (runtime model attribute) ..................... 70
__endian (compiler option) .............................. 140
entry label, program ..................................... 54
enumerations, implementation-defined behavior ......... 239
enums
data representation ..................................... 156
forward declarations of ................................ 173
environment
implementation-defined behavior ...................... 236
runtime (DLIB) ........................................... 43

environment variables
C_INCLUDE ............................................. 122
QCCRX ................................................. 122
EQU (assembler directive) ................................ 152
errno.h (library header file) ........................... 217
error messages ......................................... 125
classifying for compiler ................................ 136
erro r return codes ..................................... 124
__exchange (intrinsic function) ......................... 244
_exit (library function) ................................ 56
_exit (library function) ................................ 56
implementation-defined behavior ...................... 218
__exit (library function) ............................... 56
export keyword, missing from Extended EC++ ......... 96
extended command line file
for compiler ........................................... 141
Extended Embedded C++ ................................ 92
enabling ................................................ 140
standard template library (STL) ......................... 219
extended keywords ...................................... 177
enabling (-e) .......................................... 139
overview .............................................. 9
summary ............................................... 181
syntax .................................................. 15
object attributes ....................................... 180
type attributes on data objects ......................... 178
type attributes on data pointers ......................... 179
type attributes on function pointers .................... 180
type attributes on functions ........................... 179
__code32 (function pointer) ............................ 160
__data32 (data pointer) ................................ 160
extern "C" linkage ..................................... 96
placing in memory ........................................... 105, 107
recursive
  avoiding ............................................. 114
  storing data on stack .............................. 18
reentrancy (DLIB) ................................. 216
related extensions .................................. 21
return values from ......................... 83
special function types ......................... 23
  verifying linked result ...................... 40

g
getenv (library function), configuring support for ........ 64
getzone (library function), configuring support for ....... 65
getzone.c .............................................. 65
__get_FINTV_register (intrinsic function) ............. 205
__get_interrupt_level (intrinsic function) .............. 205
__get_interrupt_state (intrinsic function) ............. 205
__get_interrupt_table (intrinsic function) ............. 206
global variables, initializing ....................... 35
guidelines, reading ................................... xxi

h
Harbison, Samuel P. .................................. xxiv
hardware support in compiler ....................... 44
hash_map (STL header file) ....................... 219
hash_set (STL header file) ....................... 219
hdrstop (pragma directive) ...................... 241
header files
  C .................................................... 217
  C++ ............................................... 218
  EC++ .............................................. 218
  library ........................................... 215
  special function registers ................... 117
  STL .............................................. 219
  DLib_defaults.h ................................ 53
dlrx/liname.h ...................................... 53
  intrinsics.h .................................... 203
  stdbool.h ....................................... 156, 217
  stddef.h ....................................... 157
  --header_context (compiler option) .......... 142
heap
  dynamic memory .................................. 19
  segments for ................................... 38
  storing data ................................... 11
  heap segment .................................. 232
  placing .......................................... 38
  heap size
    and standard I/O .................................. 38
    changing default .................................. 38
  HEAP (segment) ................................ 38, 232
  hints, optimization ............................. 113
  Huge (code model) ............................. 22
  HUGE_ValF, C99 extension ..................... 221
  HUGE_VAll, C99 extension ..................... 221

i
-I (compiler option) ................................ 142
IAR Command Line Build Utility ............... 52
IAR Systems Technical Support .................. 126
iarbuild.exe (utility) ......................... 52
iarcmp.r54 (floating-point library) ............. 44
  __IAR_SYSTEMS_ICC__ (predefined symbol) .... 211
  __ICCRX__ (predefined symbol) .............. 211
  icons, in this guide ............................. xxv
  identifiers, implementation-defined behavior .... 236
  IEEE format, floating-point values ............ 158
  __illegal_opcode (intrinsic function) ...... 206
  implementation-defined behavior .............. 235
  important_typedef (pragma directive) ...... 241
  include files
    including before source files ............... 151
    specifying ................................... 122
  include_alias (pragma directive) .......... 194
  infinity ....................................... 159
  INFINITY, C99 extension ..................... 221
isunordered, C99 extension ........................................... 221
iswblank, C99 extension ........................................... 222
italic style, in this guide ........................................... xxiv-xxv
iterator (STL header file) ........................................ 219
I/O debugging, support for ...................................... 66
I/O module, overriding in runtime library .................... 51

K
keep_definition (pragma directive) .............................. 241
Kernighan & Ritchie function declarations .................... 115
disallowing .................................................................. 153
Kernighan, Brian W. .................................................. xxiv
keywords, extended ................................................... 9
Kernighan & Ritchie .................................................. 91

L
-I (compiler option) .................................................. 77, 142
labels ................................................................. 175
assembler, making public ....................................... 152
__program_start .................................................... 54
Labrosse, Jean J. ...................................................... xxiv
Lajoie, Josée .......................................................... xxiv
language extensions
  descriptions ......................................................... 167
  Embedded C++ .................................................. 91
  enabling .......................................................... 195
  enabling (-e) .................................................... 139
language overview .................................................. 3
language (pragma directive) ..................................... 195
libraries
  building DLIB ................................................... 45
  definition of ................................................... 4
  runtime .......................................................... 46
  standard template library .................................. 219
library configuration files
  DLIB .............................................................. 45
  Dlib Defaults.h ................................................ 53
dlrx/libname.h .................................................... 53
  modifying ....................................................... 53
  specifying ....................................................... 138
library documentation ........................................... 215
library features, missing from Embedded C++ ............. 92
library functions ................................................ 215
  reference information .................................... xxiii
  summary, DLIB ................................................ 217
library header files ............................................. 215
library modules
  creating ......................................................... 143
  overriding ..................................................... 51
library object files ............................................. 215
library options, setting ....................................... 8
library project template ....................................... 7, 52
-library_module (compiler option) ......................... 143
lightbulb icon, in this guide ................................. xxv
limits.h (library header file) .................................. 217
__LINE__ (predefined symbol) ................................. 211
linkage, C and C++ ............................................... 80
linker command file ........................................... 30
  customizing ................................................... 31
  using the -P command .................................... 32
  using the -Z command .................................... 32
linker map file ................................................... 41
linker segment. See segment linking
  from the command line .................................... 5
  required input .............................................. 5
Lippman, Stanley B. ............................................. xxiv
list (STL header file) ........................................... 219
listing, generating ............................................ 142
literals, compound ............................................. 171
literature, recommended ..................................... xxiii
  __LITTLE_ENDIAN__ (predefined symbol) ............... 211
llabs, C99 extension ........................................... 222
lldiv, C99 extension ........................................... 222
local variables, See auto variables...
module consistency ........................................... 68
rtmodel ...................................................... 200
module map, in linker map file .......................... 41
module name, specifying .................................. 144
module summary, in linker map file ................. 41
--module_name (compiler option) .................. 144
module_name (pragma directive) .................... 242
__monitor (extended keyword) ....................... 95
monitor functions ......................................... 24, 184
mutable attribute, in Extended EC++ .................. 92, 97

N
names block (call frame information) .................. 88
namespace support
in Extended EC++ ........................................... 92, 97
missing from Embedded C++ ............................... 92
naming conventions ........................................ xxv
NAN, C99 extension .......................................... 221
NDEBUG (preprocessor symbol) ......................... 7
new.h (library header file) ............................... 218
new.h (library header file) ............................... 218
NMIVEC (segment) ......................................... 233
non-initialized variables, hints for .................... 118
non-scalar parameters, avoiding ....................... 114
NOP (assembler instruction) .............................. 206
__noreturn (extended keyword) ......................... 185
Normal DLIB (library configuration) ................. 45
Not a number (NaN) ......................................... 159
--no_clustering (compiler option) .................. 145
--no_code_motion (compiler option) ............... 145
--no_cse (compiler option) ............................ 146
__no_init (extended keyword) ......................... 118, 185
--no_inline (compiler option) ....................... 146
__no_operation (intrinsic function) ................ 206
--no_path_in_file_macros (compiler option) ........ 147
no_pch (pragma directive) ............................... 242
--no_typedefs_in_diagnostics (compiler option) .... 147
--no_unroll (compiler option) ....................... 148
--no_warnings (compiler option) .................... 148
--no_wrap_diagnostics (compiler option) .......... 149
NULL (macro), implementation-defined behavior .... 242
numeric (STL header file) ............................... 219

O
-O (compiler option) ................................... 149
-o (compiler option) ..................................... 150
object attributes ......................................... 180
object filename
specifying in compiler ................................... 150
object module name, specifying ....................... 144
object_attribute (pragma directive) ............... 118, 197
--omit_types (compiler option) ..................... 150
once (pragma directive) ................................ 242
--only_stdout (compiler option) .................... 150
__open (library function) ............................... 61
operators
See also @ (operator)
__memory_of .............................................. 95
optimization
clustering, disabling ...................................... 145
code motion, disabling .................................. 145
common sub-expression elimination, disabling ..... 146
configuration .............................................. 7
disabling .................................................. 110
function inlining, disabling (--no_inline) ....... 146
hints ....................................................... 113
loop unrolling, disabling ............................ 148
specifying (-O) ........................................... 149
summary .................................................... 109
techniques ............................................... 110
type-based alias analysis (compiler option) ....... 112
disabling .................................................. 147
program entry label ...........................................54
programming hints ..........................................113
__program_start (label) ........................................54
projects, basic settings for ..................................5
prototypes, enforcing ...........................................153
PSW (register) ..................................................204
ptrdiff_t (integer type) ....................................161
PUBLIC (assembler directive) ...............................152
publication date, of this guide .................................ii
--public_equ (compiler option) .............................152
public_equ (pragma directive) ..............................242
putenv (library function), absent from DLIB ...........64
Q
QCCRX (environment variable) ............................122
qualifiers, implementation-defined behavior ............240
queue (STL header file) .....................................219
R
-r (compiler option) ..........................................152
raise (library function), configuring support for ....65
raise.c .........................................................65
RAM
non-zero initialized variables ..............................35
saving memory ...............................................114
range errors, in linker ........................................40
__read (library function) ....................................61
customizing ....................................................58
read formatter, selecting ....................................51
reading guidelines ..........................................xxi
reading, recommended ......................................xxiii
realloc (library function)
implementation-defined behavior ..........................244
See also heap ..................................................19
recursive functions
avoiding .......................................................114
storing data on stack ........................................18
reentrancy (DLIB) ...........................................216
reference information, typographic convention ........xxv
register parameters .........................................81–82
registered trademarks ........................................ii
registers
assigning to parameters ......................................82
callee-save, stored on stack ...............................18
FINITV
getting the value of (__FINITV_register) ..............205
writing a value to (__FINITV_register) .................207
for function returns ..........................................83
implementation-defined behavior .........................239
in assembler-level routines ................................79
INTBO
getting the value of (__interrupt_table) ...............206
writing a value to (__interrupt_table) .................208
preserved ......................................................81
scratch .........................................................81
reinterpret_cast (cast operator) .........................92
remark (diagnostic message)
classifying for compiler ....................................136
enabling in compiler .........................................153
--remarks (compiler option) .............................153
remarks (diagnostic message) .............................125
remove (library function) ...................................61
implementation-defined behavior .........................243
rename (library function) ...................................61
implementation-defined behavior .........................243
__ReportAssert (library function) .........................66
required (pragma directive) ..............................199
--require_prototypes (compiler option) ............153
return addresses ............................................84
return values, from functions ............................83
Ritchie, Dennis M. ...........................................xxiv
__RMPA_B (intrinsic function) .............................206
__RMPA_L (intrinsic function) .............................206
__RMPA_W (intrinsic function) .............................207
RMPA.B (assembler instruction) .........................206
RMPAL.L (assembler instruction) .........................206
Index

S

S

separate segments, declaring .......................... 201
segment (pragma directive) ............................. 201
segments .................................................. 223
code ..................................................... 39
data ....................................................... 33
definition of ............................................. 29
initialized data .......................................... 35
introduction .............................................. 29
located data .............................................. 39
naming .................................................... 34
packing in memory ........................................ 32
placing in sequence ...................................... 32
static memory ........................................... 33
summary .................................................. 233
too long for address range ............................... 40
too long in linker ........................................ 40
_segment_begin (extended operator) .................. 169
_segment_end (extended operator) ..................... 169

semaphores
C example ................................................ 24
C++ example ............................................. 26
operations on ............................................. 185
set (STL header file) ..................................... 219
setjmp.h (library header file) ........................ 217
setlocale (library function) ............................ 63
settings, basic for project configuration ............. 5
_set_FINTV_register (intrinsic function) ............ 207
_set_interrupt_level (intrinsic function) ............ 207
_set_interrupt_state (intrinsic function) ............ 207
_set_interrupt_table (intrinsic function) .......... 208
severity level, of diagnostic messages ............... 125
specifying .............................................. 126

SFR
accessing special function registers .................... 117
declaring extern special function registers .......... 107
shared object ............................................ 124
short (data type) ....................................... 156
signal (library function)
configuring support for ................................ 65
implementation-defined behavior
signal.c
signal.h (library header file)
signbit, C99 extension
signed char (data type)
specifying
signed int (data type)
signed long long (data type)
signed long (data type)
signed short (data type)
--silent (compiler option)
silent operation, specifying in compiler
64-bits (floating-point format)
skeleton code, creating for assembler language interface
slist (STL header file)
__software_interrupt (intrinsic function)
source files, list all referred
special function registers (SFR)
special function types
overview
sprite (library function)
choosing formatter
sstream (library header file)
stack
advantages and problems using
changing default size of
cleaning after function return
contents of
layout
saving space
size
supervisor mode
user mode
stack parameters
stack pointer
stack pointer register, considerations
stack segments, placing in memory
stack (STL header file)
standard error, redirecting in compiler
standard input
standard output
specifying in compiler
standard template library (STL)
in Extended EC++
missing from Embedded C++
startup code
placement of
See also CSTART
startup, system
statements, implementation-defined behavior
static clustering (compiler transformation)
static data, in linker command file
static memory segments
static overlay
static variables
initialization
taking the address of
static_cast (cast operator)
std namespace, missing from Embedded C++
and Extended EC++
stdarg.h (library header file)
stddef.h (library header file)
__STDC_VERSION__ (predefined symbol)
STDC (pragma directive)
__STDC__ (predefined symbol)
stdbool.h (library header file)
stdarg.h (library header file)
and Extended EC++
static_cast (cast operator)
std namespace, missing from Embedded C++
and Extended EC++
stdarg.h (library header file)
stddef.h (library header file)
__STDC_VERSION__ (predefined symbol)
STDC (pragma directive)
__STDC__ (predefined symbol)
stdbool.h (library header file)
stdarg.h (library header file)
warnings .................................................. 125
classifying in compiler ................................. 137
disabling in compiler .................................. 148
exit code .................................................. 154
warnings (pragma directive) ........................... 242
--warnings_affect_exit_code (compiler option) .... 124, 154
--warnings_are_errors (compiler option) .......... 154
wchar_t (data type), adding support for in C ...... 157
wchar.h (library header file) ......................... 217, 220
wchar.h, added C functionality ....................... 222
wstof, C99 extension .................................. 222
wstolb, C99 extension ................................ 222
wctype.h (library header file) ....................... 217, 222
added C functionality ................................ 222
web sites, recommended ................................ xxiv
__write (library function) ....................... 61, 220
customizing ............................................. 58

X

XCFI_COMMON (call frame information macro) ...... 90
XCFI_NAMES (call frame information macro) ...... 90
XCHG (assembler instruction) ......................... 205
XLINK errors
range error ............................................ 40
segment too long ..................................... 40
XLINK output files .................................... 5
XLINK segment memory types ......................... 30
xreportassert.c ........................................ 66

Symbols

_EXIT (library function) ............................... 56
_exit (library function) .............................. 56
_Exit, C99 extension .................................. 222
_Pragma (predefined symbol) ......................... 213
_ALIGNOF__ (operator) ................................ 169
_asm (language extension) ........................... 170
_BASE_FILE__ (predefined symbol) .................. 210
_BIG_ENDIAN__ (predefined symbol) .................. 210
_BUILD_NUMBER__ (predefined symbol) ............... 204
_break (intrinsic function) .......................... 210
_calling_conv (runtime model attribute) ........... 70
_close (library function) ............................ 61
_CODE_MODEL__ (predefined symbol) ................. 210
_code24 (extended keyword) ......................... 182
_code32 (extended keyword) ......................... 182
_code32 (function pointer) ......................... 160
_core (runtime model attribute) ..................... 70
_cplusplus (predefined symbol) ..................... 210
_DATA_MODEL__ (predefined symbol) ................. 210
_data16 (extended keyword) ......................... 183
_data24 (extended keyword) ......................... 183
_data32 (data pointer) .............................. 160
_data32 (extended keyword) ......................... 183
_DATE__ (predefined symbol) ......................... 210
_delay_cycles (intrinsic function) ................. 204
_disable_interrupt (intrinsic function) .......... 204
_double_size (runtime model attribute) .......... 70
_DOUBLE__ (predefined symbol) ...................... 210
_embedded_cplusplus (predefined symbol) ......... 210
_enable_interrupt (intrinsic function) .......... 204
_endian (runtime model attribute) ................. 70
_exchange (intrinsic function) ..................... 205
__exit (library function) ............................ 56
__fast_interrupt (extended keyword) ............... 184
__FILE__ (predefined symbol) ....................... 211
__FUNCTION__ (predefined symbol) ................. 176, 211
__func__ (predefined symbol) ....................... 176, 211
gets, in stdio.h .................................... 222
__get_FINTV_register (intrinsic function) ........ 205
__get_INTERRUPT_STATE (intrinsic function) ...... 206
__get_INTERRUPT_LEVEL (intrinsic function) ...... 206
__IAR_SYSTEMS_ICC__ (predefined symbol) ...... 211
__ICCRX__ (predefined symbol) ...................... 211
__Illegal_opcode (intrinsic function) ............. 206
Numerics

32-bits (floating-point format) .............................. 158
64-bit data types, avoiding ............................... 101
64-bits (floating-point format) ......................... 158

@ (operator)
  placing at absolute address ............................. 106
  placing in segments ................................. 107
#include files, specifying .............................. 122, 142
#warning message (preprocessor extension) ........ 213