IAR C/C++ Compiler
Reference Guide

for Texas Instruments’
MSP430 Microcontroller Family
COPYRIGHT NOTICE
Copyright © 1996–2010 IAR Systems AB.

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

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

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

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Texas Instruments is a registered trademark of Texas Instruments Incorporated.

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

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

EDITION NOTICE
Eighth edition: March 2010
Part number: C430-8

This guide applies to version 5.x of IAR Embedded Workbench® for Texas Instruments’ MSP430 microcontroller family.

Internal reference: M3, 6.0.x, IJOA.
Brief contents

Tables ..................................................................................................................... xxi
Preface ................................................................................................................. xxiii

Part 1. Using the compiler ................................................................. 1
Getting started ................................................................................................. 3
Data storage ...................................................................................................... 13
Functions ............................................................................................................... 23
Placing code and data ....................................................................................... 31
The DLIB runtime environment ................................................................. 45
The CLIB runtime environment ...................................................................... 77
Assembler language interface ........................................................................... 85
Using C ................................................................................................................ 105
Using C++ .......................................................................................................... 113
Efficient coding for embedded applications ................................................... 127

Part 2. Reference information ......................................................... 145
External interface details .................................................................................. 147
Compiler options ............................................................................................... 153
Data representation ............................................................................................ 189
Extended keywords ............................................................................................ 201
Pragma directives ............................................................................................... 213
Intrinsic functions ............................................................................................... 231
The preprocessor ............................................................................................... 241
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library functions</td>
<td>247</td>
</tr>
<tr>
<td>Segment reference</td>
<td>257</td>
</tr>
<tr>
<td>Implementation-defined behavior</td>
<td>269</td>
</tr>
<tr>
<td>Index</td>
<td>285</td>
</tr>
</tbody>
</table>
Contents

Tables ..................................................................................................................... xxi
Preface ................................................................................................................. xxiii

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

Part 1. Using the compiler ........................................................... 1

Getting started ............................................................................................... 3

IAR language overview ................................................................. 3
Supported MSP430 devices .............................................................................. 4
Building applications—an overview ............................................................... 4
  Compiling ............................................................................................. 5
  Linking ................................................................................................ 5

Basic project configuration .............................................................................. 5
  Processor configuration ........................................................................... 6
  Data model (MSP430X only) ...................................................................... 7
  Size of double floating-point type .......................................................... 7
  Optimization for speed and size ............................................................... 8
  Runtime environment ............................................................................. 8
  Position-independent code ...................................................................... 10

Special support for embedded systems ...................................................... 10
  Extended keywords ................................................................................. 10
 Pragma directives .................................................................................. 11
  Predefined symbols ............................................................................ 11
  Special function types .......................................................................... 11
Accessing low-level features ............................................................. 11

Data storage .......................................................................................... 13

Introduction .......................................................................................... 13
Different ways to store data ................................................................. 13

Data models (MSP430X only) ............................................................... 14
Specifying a data model ...................................................................... 14

Memory types (MSP430X only) .......................................................... 15
Data16 ................................................................................................ 16
Data20  ............................................................................................... 16
Using data memory attributes ......................................................... 16
Pointers and memory types ............................................................. 18
Structures and memory types ......................................................... 18
More examples ................................................................................... 19

C++ and memory types (MSP430X only) ........................................ 20
Auto variables—on the stack ........................................................... 20
The stack ............................................................................................ 20

Dynamic memory on the heap .......................................................... 21

Functions .............................................................................................. 23

Function-related extensions .............................................................. 23
Primitives for interrupts, concurrency, and OS-related programming .............................................................. 23
Interrupt functions ........................................................................ 23
Monitor functions ........................................................................... 26
C++ and special function types .................................................... 30

Execution in RAM ............................................................................... 30

Placing code and data .......................................................................... 31

Segments and memory ........................................................................ 31
What is a segment? ........................................................................... 31

Placing segments in memory ............................................................ 32
Customizing the linker configuration file ........................................ 33

Data segments .................................................................................... 35
Static memory segments ................................................................. 35
The stack ................................................................. 38
The heap ................................................................. 39
Located data .......................................................... 41
User-defined segments ........................................... 41

**Code segments** .................................................. 41
Startup code .......................................................... 41
Normal code ......................................................... 42
Interrupt functions for MSP430X ......................... 42
Interrupt vectors .................................................. 42
Code in RAM ......................................................... 42

**C++ dynamic initialization** ................................. 43

**Verifying the linked result of code and data placement** 43
Segment too long errors and range errors ............... 43
Linker map file ..................................................... 44

**The DLIB runtime environment** ......................... 45

**Introduction to the runtime environment** ............ 45
Runtime environment functionality ....................... 45
Setting up the runtime environment ....................... 46

**Using a prebuilt library** ...................................... 47
Choosing a library ................................................ 47
Customizing a prebuilt library without rebuilding .... 49

**Choosing formatters for printf and scanf** .......... 50
Choosing printf formatter ..................................... 50
Choosing scanf formatter ...................................... 51

**Application debug support** ............................... 52
Including C-SPY debugging support ..................... 52
The debug library functionality ......................... 53
The C-SPY Terminal I/O window ......................... 54

**Overriding library modules** ............................... 54

**Building and using a customized library** ............ 56
Setting up a library project .................................. 56
Modifying the library functionality ..................... 57
Using a customized library .................................. 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>System startup and termination</td>
<td>57</td>
</tr>
<tr>
<td>System startup</td>
<td>58</td>
</tr>
<tr>
<td>System termination</td>
<td>60</td>
</tr>
<tr>
<td>Customizing system initialization</td>
<td>61</td>
</tr>
<tr>
<td>__low_level_init</td>
<td>61</td>
</tr>
<tr>
<td>Modifying the file csstartup.s43</td>
<td>62</td>
</tr>
<tr>
<td>Library configurations</td>
<td>62</td>
</tr>
<tr>
<td>Choosing a runtime configuration</td>
<td>62</td>
</tr>
<tr>
<td>Standard streams for input and output</td>
<td>63</td>
</tr>
<tr>
<td>Implementing low-level character input and output</td>
<td>63</td>
</tr>
<tr>
<td>Configuration symbols for printf and scanf</td>
<td>65</td>
</tr>
<tr>
<td>Customizing formatting capabilities</td>
<td>66</td>
</tr>
<tr>
<td>File input and output</td>
<td>66</td>
</tr>
<tr>
<td>Locale</td>
<td>67</td>
</tr>
<tr>
<td>Locale support in prebuilt libraries</td>
<td>67</td>
</tr>
<tr>
<td>Customizing the locale support</td>
<td>68</td>
</tr>
<tr>
<td>Changing locales at runtime</td>
<td>69</td>
</tr>
<tr>
<td>Environment interaction</td>
<td>69</td>
</tr>
<tr>
<td>The.getenv function</td>
<td>70</td>
</tr>
<tr>
<td>The.system function</td>
<td>70</td>
</tr>
<tr>
<td>Signal and raise</td>
<td>70</td>
</tr>
<tr>
<td>Time</td>
<td>71</td>
</tr>
<tr>
<td>Strtod</td>
<td>71</td>
</tr>
<tr>
<td>Pow</td>
<td>71</td>
</tr>
<tr>
<td>Assert</td>
<td>72</td>
</tr>
<tr>
<td>Heaps</td>
<td>72</td>
</tr>
<tr>
<td>Hardware multiplier support</td>
<td>72</td>
</tr>
<tr>
<td>Checking module consistency</td>
<td>73</td>
</tr>
<tr>
<td>Runtime model attributes</td>
<td>73</td>
</tr>
<tr>
<td>Using runtime model attributes</td>
<td>74</td>
</tr>
<tr>
<td>Predefined runtime attributes</td>
<td>75</td>
</tr>
<tr>
<td>User-defined runtime model attributes</td>
<td>76</td>
</tr>
</tbody>
</table>
The CLIB runtime environment .............................................................. 77
  Runtime environment ........................................................................ 77
Input and output ........................................................................................ 79
  Character-based I/O ........................................................................ 79
  Formatters used by printf and sprintf .............................................. 79
  Formatters used by scanf and sscanf ................................................. 81
System startup and termination ............................................................ 81
  System startup ............................................................................. 81
  System termination ....................................................................... 82
Overriding default library modules ..................................................... 82
Customizing system initialization ......................................................... 82
C-SPY runtime interface ..................................................................... 82
  The debugger Terminal I/O window .............................................. 83
  Termination ................................................................................ 83
Hardware multiplier support ............................................................... 83
Checking module consistency ............................................................. 83
Assembler language interface ............................................................. 85
  Mixing C and assembler ................................................................ 85
    Intrinsic functions .................................................................... 85
    Mixing C and assembler modules ............................................ 86
    Inline assembler ....................................................................... 87
Calling assembler routines from C ....................................................... 88
  Creating skeleton code ............................................................... 88
  Compiling the code ................................................................... 89
Calling assembler routines from C++ ................................................ 90
Calling convention ............................................................................. 91
  Choosing a calling convention .................................................... 92
  Function declarations ................................................................. 92
  Using C linkage in C++ source code ........................................... 93
  Preserved versus scratch registers .............................................. 93
  Function entrance .................................................................... 94
  Function exit .......................................................................... 96
  Restrictions for special function types ...................................... 97
Efficient coding for embedded applications ......................... 127

Selecting data types ................................................................. 127
  Using efficient data types ......................................................... 127
  Floating-point types ............................................................... 128
  Alignment of elements in a structure ....................................... 129
  Anonymous structs and unions ............................................... 129

Controlling data and function placement in memory .......... 131
  Data placement at an absolute location .................................. 132
  Data and function placement in segments ............................... 133

Controlling compiler optimizations ......................................... 135
  Scope for performed optimizations ........................................ 135
  Optimization levels ............................................................... 136
  Speed versus size ............................................................... 136
  Fine-tuning enabled transformations ..................................... 137

Facilitating good code generation ............................................. 139
  Writing optimization-friendly source code .............................. 139
  Saving stack space and RAM memory ..................................... 140
  Function prototypes ............................................................. 140
  Integer types and bit negation ............................................... 141
  Protecting simultaneously accessed variables ........................ 142
  Accessing special function registers ...................................... 142
  Non-initialized variables ....................................................... 144
  Efficient switch statements ................................................... 144

Part 2. Reference information .................................................. 145

External interface details ....................................................... 147

Invocation syntax ................................................................. 147
  Compiler invocation syntax .................................................. 147
  Passing options ................................................................. 147
  Environment variables ......................................................... 148

Include file search procedure .................................................. 148

Compiler output ................................................................. 149
Diagnostics ................................................................. 151
Message format ......................................................... 151
Severity levels .......................................................... 151
Setting the severity level ........................................... 152
Internal error ............................................................ 152

Compiler options ....................................................... 153

Options syntax .......................................................... 153
Types of options ....................................................... 153
Rules for specifying parameters ................................. 153

Summary of compiler options .................................... 155

Descriptions of options .............................................. 159
--c89 ................................................................. 159
--char_is_signed .................................................. 159
--char_is_unsigned .............................................. 159
--clib ............................................................... 160
--core ............................................................... 160
-\D ................................................................. 160
--data_model ...................................................... 161
--debug, -r ........................................................ 162
--dependencies .................................................... 162
--diag_error ...................................................... 163
--diag_remark ..................................................... 163
--diag_suppress .................................................. 164
--diag_warning .................................................... 164
--diagnostics_tables ............................................ 164
--discard_unused_publics ...................................... 165
--dlib ............................................................... 165
--dlib_config ..................................................... 165
--double .......................................................... 166
-\e ................................................................. 167
--ec++ ............................................................. 167
--ec++ ............................................................. 167
--enable_multibytes .............................................. 168
--error_limit ................................................................. 168
-1 ................................................................................. 168
--header_context ............................................................ 169
-1 ................................................................................. 169
-1 ................................................................................. 169
--library_module ......................................................... 170
--lock_r4 ....................................................................... 171
--lock_r5 ....................................................................... 171
--mfc ............................................................................. 171
--migration_preprocessor_extensions ............................ 172
--module_name ............................................................. 172
--multiplier ................................................................. 173
--multiplier_location .................................................... 173
--no_code_motion .......................................................... 174
--no_cse ........................................................................ 174
--no_inline ..................................................................... 174
--no_path_in_file_macros .............................................. 175
--no_static_destruction ................................................ 175
--no_system_include ..................................................... 175
--no_tbaa ...................................................................... 176
--no_typedefs_in_diagnostics ........................................ 176
--no_unroll ..................................................................... 177
--no_warnings ............................................................... 177
--no_wrap_diagnostics .................................................. 177
-O .................................................................................. 178
--omit_types ................................................................. 178
--only_stdout ............................................................... 179
--output, -o ................................................................. 179
--pic ............................................................................. 179
--predef_macros ............................................................ 179
--preinclude .................................................................. 180
--preprocess ................................................................. 180
--public_equ ................................................................. 181
--reduce_stack_usage .................................................... 181
--regvar_r4 ................................................................. 181
--regvar_r5 ................................................................. 182
--relaxed_fp ............................................................... 182
--remarks ................................................................. 183
--require_prototypes .................................................. 183
--save_reg20 .............................................................. 183
--segment ................................................................. 184
--silent ................................................................... 185
--strict .................................................................. 185
--system_include_dir ............................................... 186
--vla ................................................................. 186
--warnings_affect_exit_code .................................... 186
--warnings_are_errors ............................................ 187
--save_reg20 .............................................................. 183
--segment ................................................................. 184
--silent ................................................................... 185
--strict .................................................................. 185

Data representation ....................................................... 189

Alignment ................................................................. 189
  Alignment on the MSP430 microcontroller .................. 189

Basic data types .......................................................... 190
  Integer types ........................................................... 190
  Floating-point types ............................................... 193

Pointer types ............................................................. 195
  Function pointers .................................................... 195
  Data pointers .......................................................... 195
  Casting .................................................................. 195

Structure types ............................................................ 196
  Alignment ................................................................. 197
  General layout ........................................................ 197
  Packed structure types ............................................ 197

Type qualifiers .............................................................. 199
  Declaring objects volatile ....................................... 199
  Declaring objects volatile and const ....................... 200
  Declaring objects const ......................................... 200

Data types in C++ .......................................................... 200
Extended keywords ................................................................. 201

General syntax rules for extended keywords ......................... 201
Type attributes ................................................................. 201
Object attributes ............................................................... 204

Summary of extended keywords ............................................ 204

Descriptions of extended keywords ...................................... 205
__cc_version1 ................................................................. 205
__cc_version2 ................................................................. 206
__data16 ................................................................. 206
__data20 ................................................................. 206
__interrupt ................................................................. 207
__intrinsic ................................................................. 207
__monitor ................................................................. 207
__no_init ................................................................. 208
__noreturn ................................................................. 208
__persistent ................................................................. 208
__ramfunc ................................................................. 209
__raw ................................................................. 209
__regvar ................................................................. 209
__root ................................................................. 210
__ro_placement ............................................................ 210
__save_reg20 ............................................................. 211
__task ................................................................. 211

Pragma directives ............................................................... 213

Summary of pragma directives ............................................... 213

Descriptions of pragma directives ....................................... 215
basic_template_matching ............................................... 215
bis_nmi_ie1 ............................................................... 215
bitfields ................................................................. 216
constseg ............................................................... 216
data_alignment ............................................................. 217
dataseg ................................................................. 217
diag_default .............................................................. 218
Intrinsic functions ................................................................. 231

Summary of intrinsic functions ............................................. 231

Descriptions of intrinsic functions ...................................... 232
  __bcd_add_type ................................................................. 232
  __bic_SR_register .......................................................... 233
  __bic_SR_register_on_exit .............................................. 233
  __bis_SR_register .......................................................... 234
  __bis_SR_register_on_exit .............................................. 234
  __data16_read_addr ....................................................... 234
The preprocessor ................................................................. 241

   Overview of the preprocessor .............................................. 241

   Descriptions of predefined preprocessor symbols ............... 242

   Descriptions of miscellaneous preprocessor extensions ... 244

   NDEBUG ................................................................. 244

   #warning message .................................................... 245

Library functions ............................................................. 247

   Library overview ........................................................ 247

   Header files ............................................................... 247

   Library object files .................................................... 248

   Reentrancy ............................................................... 248
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR DLIB Library</td>
<td>249</td>
</tr>
<tr>
<td>C header files</td>
<td>249</td>
</tr>
<tr>
<td>C++ header files</td>
<td>250</td>
</tr>
<tr>
<td>Library functions as intrinsic functions</td>
<td>253</td>
</tr>
<tr>
<td>Added C functionality</td>
<td>253</td>
</tr>
<tr>
<td>IAR CLIB Library</td>
<td>254</td>
</tr>
<tr>
<td>Library definitions summary</td>
<td>254</td>
</tr>
<tr>
<td>Segment reference</td>
<td>257</td>
</tr>
<tr>
<td>Summary of segments</td>
<td>257</td>
</tr>
<tr>
<td>Descriptions of segments</td>
<td>258</td>
</tr>
<tr>
<td>CHECKSUM</td>
<td>259</td>
</tr>
<tr>
<td>CODE</td>
<td>259</td>
</tr>
<tr>
<td>CODE_1</td>
<td>259</td>
</tr>
<tr>
<td>CODE_ID</td>
<td>260</td>
</tr>
<tr>
<td>CSTACK</td>
<td>260</td>
</tr>
<tr>
<td>CSTART</td>
<td>260</td>
</tr>
<tr>
<td>DATA16_AC</td>
<td>261</td>
</tr>
<tr>
<td>DATA16_AN</td>
<td>261</td>
</tr>
<tr>
<td>DATA16_C</td>
<td>261</td>
</tr>
<tr>
<td>DATA16_HEAP</td>
<td>261</td>
</tr>
<tr>
<td>DATA16_1</td>
<td>262</td>
</tr>
<tr>
<td>DATA16_ID</td>
<td>262</td>
</tr>
<tr>
<td>DATA16_N</td>
<td>262</td>
</tr>
<tr>
<td>DATA16_P</td>
<td>263</td>
</tr>
<tr>
<td>DATA16_Z</td>
<td>263</td>
</tr>
<tr>
<td>DATA20_AC</td>
<td>263</td>
</tr>
<tr>
<td>DATA20_AN</td>
<td>264</td>
</tr>
<tr>
<td>DATA20_C</td>
<td>264</td>
</tr>
<tr>
<td>DATA20_HEAP</td>
<td>264</td>
</tr>
<tr>
<td>DATA20_1</td>
<td>264</td>
</tr>
<tr>
<td>DATA20_ID</td>
<td>265</td>
</tr>
<tr>
<td>DATA20_N</td>
<td>265</td>
</tr>
<tr>
<td>DATA20_P</td>
<td>265</td>
</tr>
</tbody>
</table>
DATA20_Z .......................................................... 266
DIFUNCT ............................................................................ 266
INFO .......................................................... 266
INFOA .......................................................... 267
INFOB .......................................................... 267
INFOC .......................................................... 267
INFOD .......................................................... 267
INTVEC .......................................................... 268
ISR_CODE .......................................................... 268
REGVAR_AN .......................................................... 268
RESET .......................................................... 268

Implementation-defined behavior ................................................ 269

Descriptions of implementation-defined behavior ............... 269

J.3.1 Translation .......................................................... 269
J.3.2 Environment .......................................................... 270
J.3.3 Identifiers .......................................................... 271
J.3.4 Characters .......................................................... 271
J.3.5 Integers .......................................................... 273
J.3.6 Floating point .......................................................... 273
J.3.7 Arrays and pointers .......................................................... 274
J.3.8 Hints .......................................................... 274
J.3.9 Structures, unions, enumerations, and bitfields ............ 275
J.3.10 Qualifiers .......................................................... 275
J.3.11 Preprocessing directives .......................................................... 276
J.3.12 Library functions .......................................................... 277
J.3.13 Architecture .......................................................... 282
J.4 Locale .......................................................... 282

Index ........................................................................ 285
# Tables

1: Typographic conventions used in this guide ..................................................... xxvii
2: Naming conventions used in this guide .......................................................... xxviii
3: Data model characteristics .............................................................................. 15
4: Memory types and their corresponding memory attributes ......................... 17
5: XLINK segment memory types .................................................................... 32
6: Memory layout of a target system (example) ................................................. 33
7: Memory types with corresponding segment groups ....................................... 36
8: Segment name suffixes .................................................................................. 36
9: Prebuilt libraries ........................................................................................... 48
10: Customizable items ..................................................................................... 49
11: Formatters for printf .................................................................................. 50
12: Formatters for scanf ................................................................................... 51
13: Levels of debugging support in runtime libraries ....................................... 52
14: Functions with special meanings when linked with debug library ............. 53
15: Library configurations ................................................................................ 62
16: Descriptions of printf configuration symbols ........................................... 65
17: Descriptions of scanf configuration symbols ............................................. 66
18: Low-level I/O files ...................................................................................... 67
19: Heaps and memory types ............................................................................ 72
20: Additional linker configuration files for the hardware multiplier ............ 73
21: Example of runtime model attributes ......................................................... 74
22: Predefined runtime model attributes ......................................................... 75
23: CLIB runtime libraries ............................................................................... 78
24: Registers used for passing parameters ....................................................... 95
25: Registers used for returning values ............................................................ 96
26: Call frame information resources defined in a names block .................... 100
27: Language extensions .................................................................................. 107
28: Compiler optimization levels ..................................................................... 136
29: Compiler environment variables ............................................................... 148
30: Error return codes ..................................................................................... 150
31: Compiler options summary ....................................................................... 155
Preface

Welcome to the IAR C/C++ Compiler Reference Guide for MSP430. 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 MSP430 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 MSP430 microcontroller. Refer to the documentation from Texas Instruments for information about the MSP430 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 MSP430, 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 for MSP430 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 configuration file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file cstartup, how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how to customize them. The chapter also describes system initialization and introduces the file cstartup.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

**Part 2. Reference information**

- *External interface details* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler’s diagnostic system works.
- *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.

Extended keywords gives reference information about each of the MSP430-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 MSP430-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 MSP430 microcontroller is described in a series of guides. For information about:

- Getting started using IAR Embedded Workbench and the tools it provides, refer to the guide *Getting Started with IAR Embedded Workbench®*.
- System requirements and information about how to install and register the IAR Systems products, refer to the booklet *Quick Reference* (available in the product box) and the *Installation and Licensing Guide*.
- Using the IDE and the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE for MSP430 User Guide*
- Programming for the IAR Assembler for MSP430, refer to the *MSP430 IAR Assembler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library functions, refer to the online help system
- Using the IAR CLIB Library functions, refer to the *IAR C Library Functions Reference Guide*, available from the online help system.
- Porting application code and projects created with a previous IAR Embedded Workbench for MSP430, refer to the *IAR Embedded Workbench® Migration Guide for MSP430*
Document conventions

- Using the MISRA-C:1998 rules or the MISRA-C:2004 rules, refer to the *IAR Embedded Workbench® MISRA C Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*, respectively.

All of these guides are delivered in hypertext PDF or HTML format on the installation media.

FURTHER READING

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

- *MSP430xxxx User’s Guide* provided by Texas Instruments
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]

We recommend that you visit these web sites:

- The Texas Instruments web site, [www.ti.com](http://www.ti.com), contains information and news about the MSP430 microcontrollers.
- The IAR Systems web site, [www.iar.com](http://www.iar.com), holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, [www.caravan.net/ec2plus](http://www.caravan.net/ec2plus), contains information about the Embedded C++ standard.

Document conventions

When, in this text, we refer to the programming language C, the text also applies to C++, unless otherwise stated.
Unless otherwise stated, all references to the MSP430 microcontroller refer to both the MSP430 and the MSP430X microcontroller.

When referring to a directory in your product installation, for example 430\doc, the full path to the location is assumed, for example c:\Program Files\IAR Systems\Embedded Workbench 6.\n\430\doc.

**TYPOGRAPHIC CONVENTIONS**

This guide uses the following typographic conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>computer</strong></td>
<td>• Source code examples and file paths.</td>
</tr>
<tr>
<td></td>
<td>• Text on the command line.</td>
</tr>
<tr>
<td></td>
<td>• Binary, hexadecimal, and octal numbers.</td>
</tr>
<tr>
<td><strong>parameter</strong></td>
<td>A placeholder for an actual value used as a parameter, for example</td>
</tr>
<tr>
<td></td>
<td>filename.h where filename represents the name of the file.</td>
</tr>
<tr>
<td><strong>[option]</strong></td>
<td>An optional part of a command.</td>
</tr>
<tr>
<td>**a</td>
<td>b</td>
</tr>
<tr>
<td>**(a</td>
<td>b</td>
</tr>
<tr>
<td><strong>bold</strong></td>
<td>Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.</td>
</tr>
<tr>
<td><strong>italic</strong></td>
<td>• A cross-reference within this guide or to another guide.</td>
</tr>
<tr>
<td></td>
<td>• Emphasis.</td>
</tr>
<tr>
<td>...</td>
<td>An ellipsis indicates that the previous item can be repeated an arbitrary number of times.</td>
</tr>
</tbody>
</table>

Identifies instructions specific to the IAR Embedded Workbench® IDE interface.

Identifies instructions specific to the command line interface.

Identifies helpful tips and programming hints.

Identifies warnings.

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

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

<table>
<thead>
<tr>
<th>Brand name</th>
<th>Generic term</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAR Embedded Workbench® for MSP430</td>
<td>IAR Embedded Workbench®</td>
</tr>
<tr>
<td>IAR Embedded Workbench® IDE for MSP430</td>
<td>the IDE</td>
</tr>
<tr>
<td>IAR C-SPY® Debugger for MSP430</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 MSP430</td>
<td>the compiler</td>
</tr>
<tr>
<td>IAR Assembler™ for MSP430</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>
<tr>
<td>IAR CLIB Library™</td>
<td>the CLIB 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 MSP430 includes these chapters:

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

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

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

Next, the compiler is introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the MSP430 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 MSP430:

- C, the most widely used high-level programming language in the embedded systems industry. You can build freestanding applications that follow either one of the following standards:
  - The standard ISO/IEC 9899:1999 (including technical corrigendum No.3), also known as C99. Hereafter, this standard is referred to as Standard C in this guide.
  - The standard ISO 9899:1990 (including all technical corrigenda and addendum), also known as C94, C90, C89, and ANSI C. Hereafter, this standard is referred to as C89 in this guide. This standard is required when MISRA C is enabled.

- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. 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 information about C, see the chapter *Using C*.

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

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *MSP430 IAR Assembler Reference Guide*.

---

**Supported MSP430 devices**

The IAR C/C++ Compiler for MSP430 supports all devices based on the standard Texas Instruments MSP430 microcontroller, which includes both the MSP430 architecture and the MSP430X architecture. In addition, the application can use the hardware multiplier if available on the device.

*Note:* Unless otherwise stated, in this guide all references to the MSP430 microcontroller refer to both the MSP430 and the MSP430X microcontroller.

---

**Building applications—an overview**

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

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

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

Below, the process for building an application on the command line is described. For information about how to build an application using the IDE, see the *IAR Embedded Workbench® IDE for MSP430 User Guide*.
COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r43` using the default settings:

```
icc430 myfile.c
```

You must also specify some critical options, see Basic 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 configuration file that describes the placement of code and data into the memory of the target system
- Information about the output format.

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

```
xlink myfile.r43 myfile2.r43 -s __program_start -f lnk430.xcl cl430f.r43 -o aout.a43 -r
```

In this example, `myfile.r43` and `myfile2.r43` are object files, `lnk430.xcl` is the linker configuration file, and `cl430f.r43` 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 `msp430-txt`.)

Basic project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler and linker generate the best code for the MSP430 device you are using. You can specify the options either from the command line interface or in the IDE.
You must make settings for:

- Processor configuration, that includes to select core for either the MSP430 or the MSP430X architecture, and whether there is a hardware multiplier or not
- Data model
- Size of double floating-point type
- Optimization settings
- Runtime environment
- Normal or position-independent code (not supported by the MSP430X architecture).

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 for MSP430 User Guide, respectively.

**PROCESSOR CONFIGURATION**

To make the compiler generate optimum code, you should configure it for the MSP430 microcontroller you are using.

**Core**

The IAR C/C++ Compiler for MSP430 supports the MSP430 microcontroller, both the MSP430 architecture that has 64 Kbytes of addressable memory and the MSP430X architecture that has the extended instruction set and 1 Mbyte of addressable memory.

In the IAR Embedded Workbench IDE, choose Project>Options and select an appropriate device from the Device drop-down list on the Target page. The core option will then be automatically selected.

**Note:** Device-specific configuration files for the XLINK linker and the C-SPY debugger will also be automatically selected.

Use the `--core={430|430X}` option to select the architecture for which the code is to be generated.

**Hardware multiplier**

Some MSP430 devices contain a hardware multiplier. The IAR C/C++ Compiler for MSP430 can generate code that accesses this unit. In addition, the runtime library contains hardware multiplier-aware modules that can be used instead of the normal arithmetical routines.
To direct the compiler to take advantage of the unit, choose **Project>Options** and select the **Target** page in the **General Options** category. Select a device, from the **Device** drop-down menu, that contains a hardware multiplier unit.

To use the hardware multiplier, enable it on the command line using the **--multiplier** (and in some cases also the **--multiplier_location**) option. You must also, in addition to the runtime library object file, extend the XLINK command line with one of these command files:

- `f multiplier.xcl`
- `f multiplier32.xcl`
- `f multiplier32_loc2.xcl`

For more information, see *Hardware multiplier support*, page 72.

**DATA MODEL (MSP430X ONLY)**

For the MSP430X architecture, there is a trade-off regarding the way memory is accessed, ranging from cheap access to small memory areas, up to more expensive access methods that can access any location.

In the IAR C/C++ Compiler for MSP430, 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. The following data models are supported:

- **The Small data model** specifies data16 as default memory type, which means the first 64 Kbytes of memory can be accessed. The only way to access the full 1-Mbyte memory range is to use intrinsic functions.
- **The Medium data model** specifies data16 as default memory type, which means data objects by default are placed in the first 64 Kbytes of memory. If required, the entire 1 Mbyte of memory can be accessed.
- **The Large data model** specifies data20 as default memory type, which means the entire memory can be accessed.

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

**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.
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.

Two different sets of runtime libraries are provided:

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

- The IAR CLIB Library is a light-weight library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++. Note that the legacy IAR CLIB Library is provided for backward compatibility and should not be used for new application projects.

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

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

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

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, etc. See *Library configurations*, page 62, for more information.

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

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

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

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

```
dl430fn.r43
```

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

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

```
-I msp430\inc
```

For a list of all prebuilt library object files for the IAR DLIB Library, see Table 9, *Prebuilt libraries*, page 48. Make sure to use the object file that matches your other project options.

For a list of all prebuilt object files for the IAR CLIB Library, see Table 23, *CLIB runtime libraries*, page 78. 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 50 (DLIB) and Input and output, page 79 (CLIB).
- The size of the stack and the heap, see The stack, page 38, and The heap, page 39, respectively.

POSITION-INDEPENDENT CODE

Most applications are designed to be placed at a fixed position in memory. However, by enabling the compiler option `--pic` and choosing a dedicated runtime library object file, you will enable support for a feature known as position-independent code, that allows the application to be placed anywhere in memory. This is useful, for example, when developing modules that should be loaded dynamically at runtime. See `--pic`, page 179.

The drawback of position-independent code is that the size of the code will be somewhat larger, and that interrupt vectors cannot be specified directly. Also note that global data is not position-independent.

Note: Position-independent code is not supported for the MSP430X architecture.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the MSP430 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 167 for additional information.

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

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

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

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

PREDEFINED SYMBOLS

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

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

SPECIAL FUNCTION TYPES

The special hardware features of the MSP430 microcontroller are supported by the compiler’s special function types: interrupt, monitor, task, and raw. 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 85.
Special support for embedded systems
Data storage

This chapter gives a brief introduction to the memory layout of the MSP430 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 compiler supports MSP430 devices with both the MSP430 instruction set and the MSP430X extended instruction set, which means that 64 Kbytes and 1 Mbyte of continuous memory can be used, respectively.

Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM or flash) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and memory-mapped registers for peripheral units.

The MSP430X architecture can access the lower 64 Kbyte using normal instructions and the entire memory range using more expensive extended instructions. The compiler supports this by means of memory types, where data16 memory corresponds to the lower 64 Kbytes and the data20 memory the entire 1 Mbyte memory range. To read more about this, see Memory types (MSP430X only), page 15.

Placing read-only (constant) data in data20 memory is useful for most MSP430X devices. However, using data20 memory for read/write data only makes sense if the device has RAM memory above the first 64 Kbytes.

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 or in processor registers. 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.
Data models (MSP430X only)

- Global variables, module-static variables, and local variables declared `static`.
  
  In this case, the memory is allocated once and for all. The word `static` in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see Data models (MSP430X only), page 14 and Memory types (MSP430X only), page 15.

- Dynamically allocated data.
  
  An application can allocate data on the heap, where the data remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see Dynamic memory on the heap, page 21.

Data models (MSP430X only)

Data models are only available for the MSP430X architecture.

Technically, one property that the data model specifies is the default memory type, which controls the following:

- The default placement of static and global variables, as well as constant literals
- Dynamically allocated data, for example data allocated with `malloc`, or, in C++, the operator `new`
- The default pointer type
- The placement of C++ objects like, for example, virtual functions tables.

In the Small data model, only the data16 memory type is available. In the Medium and Large data models, you can explicitly override the default memory type by using memory attributes. For information about how to specify a memory type for individual objects, see Using data memory attributes, page 16.

SPECIFYING A DATA MODEL

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

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects or pointers by explicitly specifying a memory attribute, see Using data memory attributes, page 16.
This table summarizes the different data models:

<table>
<thead>
<tr>
<th>Data model name</th>
<th>Default memory attribute</th>
<th>Data20 available</th>
<th>Placement of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>__data16</td>
<td>No</td>
<td>0–0xFFFF</td>
</tr>
<tr>
<td>Medium</td>
<td>__data16</td>
<td>Yes</td>
<td>0–0xFFFF</td>
</tr>
<tr>
<td>Large</td>
<td>__data20</td>
<td>Yes</td>
<td>0–0xFFFFF</td>
</tr>
</tbody>
</table>

Table 3: Data model characteristics

See the IAR Embedded Workbench® IDE for MSP430 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 161.

The Small data model

The Small data model can only use data16 memory, which means that all data must be placed in the lower 64 Kbytes of memory. The advantage is that the generated code can rely on the upper four bits of the processor registers never being used, which means that a saved register will occupy only two bytes, not four. Among else, this reduces the stack space needed to store preserved registers, see Preserved registers, page 94.

The Medium data model

The Medium data model uses data16 memory by default. Unlike the Small data model, the Medium data model can also use data20 memory. This data model is useful if most of the data can fit into the lower 64 Kbytes of memory, but a small number of large data structures do not.

The Large data model

In the Large data model, the data20 memory is default. This model is mainly useful for large applications with large amounts of data. Note that if you need to use the data20 memory, it is for most applications better to use the Medium data model and place individual objects in data20 memory using the __data20 memory attribute. To read more about the reasons for this, see Data20, page 16.

Memory types (MSP430X only)

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 the data16 memory access method is called data16 memory.

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

**DATA16**

The data16 memory consists of the low 64 Kbytes of memory. In hexadecimal notation, this is the address range \(0x0000-0xFFFF\).

A pointer to data16 memory is 16 bits and occupies two bytes when stored in memory. Direct accesses to data16 memory are performed using normal (non-extended) instructions. This means a smaller footprint for the application, and faster execution at run-time.

**DATA20**

Using this memory type, you can place the data objects anywhere in the entire memory range \(0x00000-0xFFFFF\). This requires the extended instructions of the MSP430X architecture, which are more expensive. Note that a pointer to data20 memory will occupy four bytes of memory, which is twice the amount needed for data16 memory.

**USING DATA MEMORY ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Keyword</th>
<th>Address range</th>
<th>Pointer size</th>
<th>Default in data model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data16</td>
<td>__data16</td>
<td>0x0-0xFFFF</td>
<td>16 bits</td>
<td>Small and Medium</td>
</tr>
<tr>
<td>Data20</td>
<td>__data20</td>
<td>0x0-0xFFFFF</td>
<td>32 bits</td>
<td>Large</td>
</tr>
</tbody>
</table>

Table 4: Memory types and their corresponding memory attributes

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 167 for additional information.

For reference information about each keyword, see Descriptions of extended keywords, page 205.

**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 201.

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

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

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

The `#pragma type_attribute` directive can also be used for specifying the memory attributes. The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Refer to the chapterPragma 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 __data20 Byte;
typedef Byte *BytePtr;
```
Memory types (MSP430X only)

Byte AByte;
BytePtr ABytePointer;

/* Defines directly */
__data20 char AByte;
char __data20 *ABytePointer;

**POINTERS AND MEMORY TYPES**

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

In the compiler, a pointer also points to some type of memory. The memory type is specified using a keyword before the asterisk. For example, a pointer that points to an integer stored in data20 memory is declared by:

```c
int __data20 * MyPtr;
```

Note that the location of the pointer variable `MyPtr` is not affected by the keyword. In the following example, however, the pointer variable `MyPtr2` is placed in data20 memory. Like `MyPtr`, `MyPtr2` points to a character in data16 memory.

```c
char __data16 * __data20 MyPtr2;
```

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

**Differences between pointer types**

A pointer must contain information needed to specify a memory location of a certain memory type. This means that the pointer sizes are different for different memory types. In the IAR C/C++ Compiler for MSP430, the size of the data16 and data20 pointers is 16 and 20 bits, respectively. However, when stored in memory they occupy two and four bytes, respectively.

In the compiler, it is illegal to convert a data20 pointer to a data16 pointer without an explicit cast.

For more information about pointers, see *Pointer types*, page 195.

**STRUCTURES AND MEMORY TYPES**

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

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

```c
struct MyStruct
{
    int mAlpha;
```
int mBeta;
};
__data20 struct MyStruct Gamma;

This declaration is incorrect:
struct MyStruct
{
    int mAlpha;
    __data20 int mBeta; /* Incorrect declaration */
};

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in data16 memory is declared. The function returns a pointer to an integer in data20 memory. It makes no difference whether the memory attribute is placed before or after the data type. To read the following examples, start from the left and add one qualifier at each step

int MyA;  
A variable defined in default memory determined by the data model in use.

int __data16 MyB;  
A variable in data16 memory.

__data20 int MyC;  
A variable in data20 memory.

int * MyD;  
A pointer stored in default memory. The pointer points to an integer in default memory.

int __data16 * MyE;  
A pointer stored in default memory. The pointer points to an integer in data16 memory.

int __data16 * __data20 MyF;  
A pointer stored in data20 memory pointing to an integer stored in data16 memory.

int __data20 * MyFunction(  
    int __data16 *);  
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 data20 memory.
C++ and memory types (MSP430X only)

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

In non-static member functions, the non-static member variables of a C++ object can be referenced via the `this` pointer, explicitly or implicitly. The `this` pointer is of the default data pointer type unless class memory is used, see Classes, page 115.

In the Small and Medium data models, this means that objects of classes with a member function can only be placed in the default memory type (`__data16`).

Static member variables can be placed individually into a data memory in the same way as free variables. For more information about C++ classes, see Classes, page 115.

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—if of course—deallocate memory on the stack.

**Advantages**

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

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

**Potential problems**

The way the stack works makes it impossible to store data that is supposed to live after the function returns. The following function demonstrates a common programming mistake. It returns a pointer to the variable x, a variable that ceases to exist when the function returns.

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

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

**Dynamic memory on the heap**

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

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

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

For MSP430X, the compiler supports heaps in both data16 and data20 memory. For more information about this, see *The heap*, page 39.
Dynamic memory on the heap

Potential problems

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

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

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

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

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

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

Function-related extensions

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

- Use primitives for interrupts, concurrency, and OS-related programming
- Facilitate function optimization
- Make functions execute in RAM
- Access hardware features.

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

For more information about optimizations, see Efficient coding for embedded applications, page 127. For information about the available intrinsic functions for accessing hardware operations, see the chapter Intrinsic functions.

Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for MSP430 provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords __interrupt, __task, __monitor, __raw, and __save_reg20
- The pragma directives #pragma vector and #pragma no_epilogue
- A number of intrinsic functions, including __enable_interrupt, __disable_interrupt, __get_interrupt_state, and __set_interrupt_state
- The compiler option --save_reg20.

INTERRUPT FUNCTIONS

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

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

The MSP430 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 MSP430 microcontroller documentation from the chip manufacturer. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors.

Interrupt vectors and the interrupt vector table

The interrupt vector is the offset into the interrupt vector table.

For the MSP430 microcontroller, the interrupt vector table typically contains 16 vectors and the table starts at the address $0xFFE0$. However, some devices contain more than 16 vectors, in which case the vectors start at a lower address. For example, if the device has 32 vectors, the table starts at address $0xFFC0$.

The interrupt vectors are placed in the segment INTVEC. The last entry in the table contains the reset vector, which is placed in a separate linker segment—RESET.

The header file io<device>.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 directive can be used. For example:

```c
#pragma vector = 0x14
__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 MSP430 microcontroller documentation for more information about the interrupt vector table.
Preventing registers from being saved at function entrance

As noted, the interrupt function preserves the content of all used processor register at the entrance and restores them at exit. However, for some very special applications, it can be desirable to prevent the registers from being saved at function entrance.

This can be accomplished by the use of the extended keyword `__raw`, for example:

```c
__raw __interrupt void my_interrupt_function()
```

This creates an interrupt service routine where you must make sure that the code within the routine does not affect any of the registers used by the interrupted environment. Typically, this is useful for applications that have an empty foreground loop and use interrupt routines to perform all work.

**Note:** The same effect can be achieved for normal functions by using the `__task` keyword.

Interrupt Vector Generator interrupt functions

The compiler provides a way to write very efficient interrupt service routines for the modules that has Interrupt Vector Generators, this includes Timer A (TAIV), Timer B (TBIV), the I²C module (I2CIV), and the ADC12 module.

The interrupt vector register contains information about the interrupt source, and the interrupt service routine normally uses a switch statement to find out which interrupt source issued the interrupt. To help the compiler generate optimal code for the switch statement, the intrinsic function `__even_in_range` can be used. This example defines a Timer A interrupt routine:

```c
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1_ISR(void)
{
    switch (__even_in_range(TAIV, 10))
    {
        case 2:  P1POUT ^= 0x04;
                  break;
        case 4:  P1POUT ^= 0x02;
                  break;
        case 10: P1POUT ^= 0x01;
                  break;
    }
}
```
The intrinsic function `__even_in_range` requires two parameters, the interrupt vector register and the last value in the allowed range, which in this example is 10. The effect of the intrinsic function is that the generated code can only handle even values within the given range, which is exactly what is required in this case as the interrupt vector register for Timer A can only be 0, 2, 4, 6, 8, or 10. If the `__even_in_range` intrinsic function is used in a case where an odd value or a value outside the given range could occur, the program will fail.

For more information about the intrinsic keyword, see `__even_in_range`, page 237.

**Interrupt functions for the MSP430X architecture**

When compiling for the MSP430X architecture, all interrupt functions are automatically placed in the segment `ISR_CODE`, which must be located in the lower 64 Kbytes of memory. If you are using one of the linker configuration files for an MSP430X device that are delivered with the product, the segment will be correctly located.

In the Small data model, all functions save only 16 bits of the 20-bit registers on entry and exit. If you have assembler routines that use the upper 4 bits of the registers, you must use either the `__save_reg20` keyword on all your interrupt functions, alternatively the `--save_reg20` compiler option.

This will ensure that the interrupt functions will save and restore all 20 bits of any 20-bit registers that are used. The drawback is that the entry and leave sequences will become slower and consume more stack space.

**Note:** If a `__save_reg20` function, compiled using either the `--lock_R4` or the `--lock_R5` option, calls another function that is not `__save_reg20` declared and does not lock R4/R5, the upper four bits of R4/R5 might be destroyed. For this reason, it is not recommended to use different settings of the `--lock_R4/R5` option for different modules.

**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 207.

Avoid using the `__monitor` keyword on large functions, since the interrupt will otherwise be turned off for a significant period of time.
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.

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

/* Function to test whether the lock is open, and if so take it.
* Returns 1 on success and 0 on failure. */
__monitor int TryGetLock(void)
{
    if (sTheLock == 0)
    {
        /* Success, nobody has the lock. */
        sTheLock = 1;
        return 1;
    }
    else
    {
        /* Failure, someone else has the lock. */
        return 0;
    }
}

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

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

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

Example of implementing a semaphore in C++

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

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

#include <intrinsics.h>

/* Class for controlling critical blocks. */
class Mutex
{
public:
    Mutex()
    {
        // Get hold of current interrupt state.
        mState = __get_interrupt_state();

        // Disable all interrupts.
        __disable_interrupt();
    }

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

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

// Enter a critical block.
{
Mutex m;

// 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, interrupt
member functions must be static. When a non-static member function is called, it must
be applied to an object. When an interrupt occurs and the interrupt function is called,
there is no object available to apply the member function to.

Execution in RAM

The __ramfunc keyword makes a function execute in RAM. The function is copied
from ROM to RAM by cstartup, see System startup and termination, page 57.

The keyword is specified before the return type:

__ramfunc void foo(void);

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

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

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

Segments and memory

In an embedded system, there 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 configuration file. Ready-made linker configuration 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 5: XLINK segment memory types*

XLINK supports several other segment memory types than the ones described above. However, they exist to support other types of 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 configuration file that contains command line options which specify the locations where the segments can be placed, thereby assuring that your application fits on the target chip. To use the same source code with different derivatives, just rebuild the code with the appropriate linker configuration file.

In particular, the linker configuration file specifies:

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

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

The config directory contains ready-made linker configuration files for all supported devices (filename extension xcl). The files contain the information required by the linker, and are ready to be used. The only change you will normally have to make to the supplied linker configuration file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you must add details about the external RAM memory area.

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

<table>
<thead>
<tr>
<th>Range</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0200–0x09FF</td>
<td>RAM</td>
</tr>
<tr>
<td>0x1000–0x10FF</td>
<td>ROM</td>
</tr>
<tr>
<td>0x1100–0xFFFF</td>
<td>RAM</td>
</tr>
</tbody>
</table>

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

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

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 configuration file

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

- The CPU used:
  ```
  -cmsp430
  ```
  This specifies your target microcontroller. (It is the same for both the MSP430 and the MSP430X architecture.)

- Definitions of constants used in the file. These are defined using the XLINK option `-D`.

- The placement directives (the largest part of the linker configuration file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

In the linker configuration file, all numbers are specified in hexadecimal format. However, neither the prefix 0x nor the suffix h is used.
Note: The supplied linker configuration 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.

-2 (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.

-2 (CONST) MYSEGMENTA=2000-CFFF
-2 (CODE) MYCODE

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

-2 (CONST) MYSMALLSEGMENT=2000-20FF
-2 (CONST) MYLARGESegment=2000-CFFF

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

Using the -P command for packed placement

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

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

-P (DATA) MYDATA=0-1FFF,10000-11FFF
If your application has an additional RAM area in the memory range 0x0000-0x07FF, you can simply add that to the original definition:

```
-P(DATA)MYDATA=0-1FFF,0000-07FF
```

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 `__persistent` keyword, meaning that they are only initialized once, when the code is downloaded, and not by `cstartup`
- Variables defined with the `__no_init` keyword, meaning that they should not be initialized at all.

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

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

Segment naming

The names of the segments consist of two parts—the segment group name and a suffix—for instance, DATA16_Z. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for example DATA16 and __data16. The following table summarizes the memory types and the corresponding segment groups:

<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>0x0000-0xFFFF</td>
</tr>
<tr>
<td>Data20</td>
<td>DATA20</td>
<td>0x00000-0xFFFFF</td>
</tr>
</tbody>
</table>

Table 7: 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 32.

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>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>Non-initialized data</td>
<td>N</td>
<td>DATA</td>
</tr>
<tr>
<td>Constants</td>
<td>C</td>
<td>CONST</td>
</tr>
<tr>
<td>Persistent data</td>
<td>P</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 8: Segment name suffixes

For a list of all supported segments, see Summary of segments, page 257.
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 non-initialized variables are placed in the segment DATA16_N.
__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 0x200-0x2FF and 0x400-0x4FF
DATA16_ID 0x600-0x7FF
```

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

```
DATA16_I 0x200-0x2FF and 0x400-0x4FF
DATA16_ID 0x600-0x6FF and 0x800-0x8FF
```

Note that the gap between the ranges will also be copied.

3. Finally, global C++ objects are constructed, if any.
Data segments for static memory in the default linker configuration file

In this example, the directives for placing the segments in the linker configuration file would be:

```plaintext
// The RAM segments
-Z(DATA)DATA16_I,DATA16_Z,DATA16_N=200-9FF

// The ROM segments
-Z(CONST)DATA16_C,DATA16_ID=1100-FFDF
```

**THE STACK**

The stack is used by functions to store variables and other information that is used locally by functions, as described in the chapter Data storage. It is a continuous block of memory pointed to by the processor stack pointer register \( SP \).

The data segment used for holding the stack is called \( \text{CSTACK} \). The system startup code initializes the stack pointer to the end of the stack segment.

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

Stack size allocation in the IDE

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

Add the required stack size in the Stack size text box.

Stack size allocation from the command line

The size of the \( \text{CSTACK} \) segment is defined in the linker configuration file.

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

```plaintext
-D_CSTACK_SIZE=size
```

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

Specify an appropriate size for your application. Note that the size is written hexadecimally without the 0x notation.
Placement of stack segment

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

-2(DATA)CSTACK+._CSTACK_SIZE#0200-09FF

Note:

- This range does not specify the size of the stack; it specifies the range of the available memory
- The # allocates the CSTACK segment at the end of the memory area. In practice, this means that the stack will get all remaining memory at the same time as it is guaranteed that it will be at least _CSTACK_SIZE bytes in size. See the IAR Linker and Library Tools Reference Guide for more information.

Stack size considerations

The compiler uses the internal data stack, CSTACK, for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too large, RAM is wasted. If the given stack size is too small, two things can happen, depending on where in memory you located your stack. Both alternatives are likely to result in application failure. Either program variables will be overwritten, leading to undefined behavior, or the stack will fall outside of the memory area, leading to an abnormal termination of your application.

For this reason, you should consider placing the stack at the end of the RAM memory.

THE HEAP

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

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

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

A heap is only included in the application if dynamic memory allocation is actually used.

Heap segments in DLIB

When using the DLIB runtime environment, you can allocate data in the default memory using the standard memory allocation functions malloc, free, etc.
However, in the Medium or Large data model using an MSP430X device, you can also allocate data in a non-default memory by using the appropriate memory attribute as a prefix to the standard functions malloc, free, calloc, and realloc, for example:

__data16_malloc

The heaps for the two memories will be placed in the segments DATA16_HEAP and DATA20_HEAP, respectively.

For information about available heaps, see Heaps, page 72.

**Heap segments in the CLIB runtime environment**

In the CLIB runtime environment one heap is available. It is placed in default memory; data16 in the Small and Medium data model (the DATA16_HEAP segment), and data20 in the Large data model (the DATA20_HEAP segment).

**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 configuration file.

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

-D_DATA16_HEAP_SIZE=size
-D_DATA20_HEAP_SIZE=size

**Note:** Normally, these lines are 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:

-2(DATA)DATA16_HEAP+DATA16_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 your DLIB runtime environment is configured to use FILE descriptors, as in the Full configuration, input and output buffers for file handling will be allocated. In that case, be aware that the size of the input and output buffers is set to 512 bytes in the stdio library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on actual hardware. 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, for example 1 Kbyte.

If you have excluded FILE descriptors from the DLIB runtime environment, as in the normal DLIB configuration, there are no input and output buffers at all.

Located Data

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

User-Defined Segments

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

Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For a complete list of all segments, see Summary of segments, page 257.

Startup Code

The segment CSTART contains code used during system startup (cstartup). The system startup code should be placed at the location where the chip starts executing code after a reset. For the MSP430 microcontroller, this is at the reset vector address. The segments must also be placed into one continuous memory space, which means that the -P segment directive cannot be used.
In the default linker configuration file, this line will place the \texttt{CSTART} segment at the address 0x1100:

\texttt{-Z(CODE)CSTART=1100-FFBF}

**NORMAL CODE**

Code for normal functions and interrupt functions is placed in the \texttt{CODE} segment. Again, this is a simple operation in the linker configuration file:

\texttt{/* For MSP430 devices */}  
\texttt{-Z(CODE)CODE=1100-FFDF}

\texttt{/* For MSP430X devices */}  
\texttt{-Z(CODE)CODE=1100-FFBF,10000-FFFFF}

**INTERRUPT FUNCTIONS FOR MSP430X**

When you compile for the MSP430X architecture, the interrupt functions are placed in the \texttt{ISR\_CODE} segment. This too is a simple operation in the linker configuration file:

\texttt{-Z(CODE)ISR\_CODE=1100-FFDF}

**INTERRUPT VECTORS**

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the segment \texttt{INTVEC}. For the MSP430 microcontroller, it should end at address 0xFFFF. For a device with 16 interrupt vectors, this means that the segment should start at the address 0xFFE0, for example:

\texttt{-Z(CONST)INTVEC=FFE0-FFFF}

The system startup code places the reset vector in the \texttt{RESET} segment; the \texttt{INTVEC} segment cannot be used by the system startup code because the size of the interrupt vector table varies between different devices. In the linker configuration file it can look like this:

\texttt{-Z(CONST)RESET=FFFE-FFFF}

An application that does not use the standard startup code can either use the \texttt{RESET} segment, or define an interrupt function on the reset vector, in which case the \texttt{INTVEC} segment is used.

**CODE IN RAM**

The segment \texttt{CODE\_I} holds code which executes in RAM and which was initialized by \texttt{CODE\_ID}.
The linker configuration file uses the XLINK option -q to specify automatic setup for copy initialization of segments. This will cause the linker to generate a new initializer segment into which it will place all data content of the code segment. Everything else, such as symbols and debugging information, will still be associated with the code segment. Code in the application must at runtime copy the contents of the initializer segment in ROM to the code segment in RAM. This is very similar to how initialized variables are treated.

```c
/* __ramfunc code copied to and executed from RAM */
-Z (DATA) CODE_I=RAMSTART-RAMEND
-Q CODE_I=CODE_ID
```

**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:

```c
-Z (CONST) DIFUNCT=1100-2000
```

For additional information, see DIFUNCT, page 266.

**Verifying the linked result of code and data placement**

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

**SEGMENT TOO LONG ERRORS AND RANGE ERRORS**

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

Some instructions do not work unless a certain condition holds after linking, for example that a branch must be within a certain distance or that an address must be even. XLINK verifies that the conditions hold when the files are linked. If a condition is not satisfied, XLINK generates a range error or warning and prints a description of the error.
For 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 `-x` 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 for MSP430 User Guide*. 
The DLIB runtime environment

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

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

Introduction to the runtime environment

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

**RUNTIME ENVIRONMENT FUNCTIONALITY**

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

The runtime library is delivered both as prebuilt libraries and (depending on your product package) as source files, and you can find them in the product subdirectories `430\lib` and `430\src\lib`, respectively.

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

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
  - Peripheral unit registers and interrupt definitions in include files
  - The MSP430 hardware multiplier peripheral unit.
  - Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
A floating-point environment (fenv) that contains floating-point arithmetics support, see fenv.h, page 253.

- Special compiler support, for instance functions for switch handling or integer arithmetics.

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

**SETTING UP THE RUNTIME ENVIRONMENT**

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

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

- Choose which library to use—the DLIB or the CLIB library
  
  Use the compiler option --clib or --dlib, respectively. For more details about the libraries, see Library overview, page 247.

- Choose which runtime library object file to use
  
  The IDE will automatically choose a runtime library based on your project settings. If you build from the command line, you must specify the object file explicitly. See Using a prebuilt library, page 47.

- Choose which predefined runtime library configuration to use—Normal or Full
  
  You can configure the level of support for certain library functionality, for example, locale, file descriptors, and multibyte characters. If you do not specify anything, a default library configuration file that matches the library object file is automatically used. To specify a library configuration explicitly, use the --dlib_config compiler option. See Library configurations, page 62.

- Optimize the size of the runtime library
  
  You can specify the formatters used by the functions printf, scanf, and their variants, see Choosing formatters for printf and scanf, page 50. You can also specify the size and placement of the stack and the heaps, see The stack, page 39, and The heap, page 39, respectively.

- Include debug support for runtime and I/O debugging
  
  The library offers C-SPY debug support and if you want to debug your application, you must choose to use it, see Application debug support, page 52

- Adapt the library functionality
  
  Some library functions must be customized to suit your target hardware, for example low-level functions for character-based I/O, environment functions, signal functions,
and time functions. This can be done without rebuilding the entire library, see
\textit{Overriding library modules}, page 54.

- Customize system initialization
  
  It is likely that you need to customize the source code for system initialization, for
  example, your application might need to initialize memory-mapped special function
  registers, or omit the default initialization of data segments. You do this by
  customizing the routine \texttt{\_\_low\_level\_init}, which is executed before the data
  segments are initialized. See \textit{System startup and termination}, page 57 and
  \textit{Customizing system initialization}, page 61.

- Configure your own library configuration files
  
  In addition to the prebuilt library configurations, you can make your own library
  configuration, but that requires that you \texttt{rebuild} the library. This gives you full
  control of the runtime environment. See \textit{Building and using a customized library},
  page 56.

- Check module consistency
  
  You can use runtime model attributes to ensure that modules are built using
  compatible settings, see \textit{Checking module consistency}, page 73.

\section*{Using a prebuilt library}

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

- Core
- Data model
- Size of the \texttt{double} floating-point type
- Library configuration—Normal or Full
- Position-independent code.

\section*{CHOOSING A LIBRARY}

The IDE will include the correct library object file and library configuration file based
on the options you select. See the \textit{IAR Embedded Workbench® IDE for MSP430 User
Guide} for additional information.

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

- Specify which library object file to use on the \texttt{XLINK} command line, for instance:
  \texttt{dl430dfp.r43}
### Using a prebuilt library

- If you do not specify a library configuration, the default will be used. However, you can specify the library configuration explicitly for the compiler:
  
  ```
  --dlib_config C:\...\dl430dfp.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 `430\lib\dlib`.

These prebuilt runtime libraries are available:

<table>
<thead>
<tr>
<th>Library</th>
<th>Core</th>
<th>Data model</th>
<th>Size of double</th>
<th>Library configuration</th>
<th>PIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>dl430fn.r43</td>
<td>430</td>
<td>—</td>
<td>32</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430fnp.r43</td>
<td>430</td>
<td>—</td>
<td>32</td>
<td>Normal</td>
<td>Yes</td>
</tr>
<tr>
<td>dl430ff.r43</td>
<td>430</td>
<td>—</td>
<td>32</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>dl430ffp.r43</td>
<td>430</td>
<td>—</td>
<td>32</td>
<td>Full</td>
<td>Yes</td>
</tr>
<tr>
<td>dl430dn.r43</td>
<td>430</td>
<td>—</td>
<td>64</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430dnp.r43</td>
<td>430</td>
<td>—</td>
<td>64</td>
<td>Normal</td>
<td>Yes</td>
</tr>
<tr>
<td>dl430df.r43</td>
<td>430</td>
<td>—</td>
<td>64</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>dl430dfp.r43</td>
<td>430</td>
<td>—</td>
<td>64</td>
<td>Full</td>
<td>Yes</td>
</tr>
<tr>
<td>dl430xsfn.r43</td>
<td>430X</td>
<td>Small</td>
<td>32</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430xsff.r43</td>
<td>430X</td>
<td>Small</td>
<td>32</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>dl430xsdn.r43</td>
<td>430X</td>
<td>Small</td>
<td>64</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430xsdf.r43</td>
<td>430X</td>
<td>Small</td>
<td>64</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>dl430xsmfn.r43</td>
<td>430X</td>
<td>Medium</td>
<td>32</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430xmff.r43</td>
<td>430X</td>
<td>Medium</td>
<td>32</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>dl430xmdn.r43</td>
<td>430X</td>
<td>Medium</td>
<td>64</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430xmdf.r43</td>
<td>430X</td>
<td>Medium</td>
<td>64</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>dl430xlfn.r43</td>
<td>430X</td>
<td>Large</td>
<td>32</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430xlff.r43</td>
<td>430X</td>
<td>Large</td>
<td>32</td>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>dl430xldn.r43</td>
<td>430X</td>
<td>Large</td>
<td>64</td>
<td>Normal</td>
<td>No</td>
</tr>
<tr>
<td>dl430xldf.r43</td>
<td>430X</td>
<td>Large</td>
<td>64</td>
<td>Full</td>
<td>No</td>
</tr>
</tbody>
</table>

*Table 9: Prebuilt libraries*
**Library filename syntax**

The names of the libraries are constructed in this way:

\[ \text{\{\text{lib}\}\{\text{core}\}\{\text{data\_model}\}\{\text{size\_of\_double}\}\{\text{lib\_config}\}\{\text{pic}\}.r43} \]

where

- \text{\textit{lib}} is \text{dl} for the IAR DLIB runtime environment
- \text{\textit{core}} is either 430 or 430x
- \text{\textit{data\_model}} is empty for MSP430 devices. For MSP430X devices it is one of \text{s}, \text{m}, or \text{l}, for the Small, Medium, and Large data model, respectively
- \text{\textit{size\_of\_double}} is either \text{f} for 32 bits or \text{d} for 64 bits
- \text{\textit{lib\_config}} is either \text{n} or \text{f} for normal and full, respectively.
- \text{\textit{pic}} is either empty for no support for position-independent code or \text{p} for position-independent code.

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

**CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING**

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

These items can be customized:

<table>
<thead>
<tr>
<th>Items that can be customized</th>
<th>Described in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatters for printf and scanf</td>
<td>Choosing formatters for printf and scanf, page 50</td>
</tr>
<tr>
<td>Startup and termination code</td>
<td>System startup and termination, page 57</td>
</tr>
<tr>
<td>Low-level input and output</td>
<td>Standard streams for input and output, page 63</td>
</tr>
<tr>
<td>File input and output</td>
<td>File input and output, page 66</td>
</tr>
<tr>
<td>Low-level environment functions</td>
<td>Environment interaction, page 69</td>
</tr>
<tr>
<td>Low-level signal functions</td>
<td>Signal and raise, page 70</td>
</tr>
<tr>
<td>Low-level time functions</td>
<td>Time, page 71</td>
</tr>
<tr>
<td>Size of heaps, stacks, and segments</td>
<td>Placing code and data, page 31</td>
</tr>
</tbody>
</table>

Table 16: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 54.
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, you can optimize these functions even further, see Configuration symbols for printf and scanf, page 65.

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

Table 11: 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 65.

**Specifying the print formatter in the IDE**

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

To use any other formatter than the default full formatter `_Printf`, add one of these lines in the linker configuration 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>Small</th>
<th>Large</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic specifiers c, d, i, o, p, s, u, X, x, and %</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Multibyte support</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>Floating-point specifiers a, and A</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Floating-point specifiers e, E, f, F, g, and G</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Conversion specifier n</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Scan set [ ] and</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Assignment suppressing *</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>long long support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 12: 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 65.

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 full formatter _Scanf, add one of these lines in the linker configuration file you are using:

- e_ScanfLarge=_Scanf
- e_ScanfSmall=_Scanf

Application debug support

In addition to the tools that generate debug information, there is a debug version of the DLIB low-level interface (typically, I/O handling and basic runtime support). If your application uses this interface, you can either use the debug version of the interface or you must implement the functionality of the parts that your application uses.

INCLUDING C-SPY DEBUGGING SUPPORT

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

This table describes the different levels of debugging support:

<table>
<thead>
<tr>
<th>Debugging support</th>
<th>Linker option in the IDE</th>
<th>Linker command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic debugging</td>
<td>Debug information for C-SPY</td>
<td>-Fubrof</td>
<td>Debug support for C-SPY without any runtime support</td>
</tr>
<tr>
<td>Runtime debugging*</td>
<td>With runtime control modules</td>
<td>-r</td>
<td>The same as -Fubrof, 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 modules</td>
<td>-rt</td>
<td>The same as -r, but also includes debugger support for I/O handling, which means that stdin and stdout are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.</td>
</tr>
</tbody>
</table>

* If you build your application project with this level of debug support, certain functions in the library are replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see The debug library functionality, page 53.
In the IDE, choose **Project>Options>Linker**. On the **Output** page, select the appropriate **Format** option.

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

**THE DEBUG LIBRARY FUNCTIONALITY**

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

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

The mechanism used for implementing this feature works as follows:

The debugger will detect the presence of the function __DebugBreak, which will be part of the application if you linked it with the XLINK 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.

If you have included the runtime library debugging support, C-SPY will make the following responses when the application uses the DLIB low-level interface:

<table>
<thead>
<tr>
<th>Function in DLIB low-level interface</th>
<th>Response by C-SPY</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>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>Closes the associated host file on the host computer</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>stdin, stdout, and stderr 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>
<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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function in DLIB low-level interface</th>
<th>Response by C-SPY</th>
</tr>
</thead>
<tbody>
<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 14: Functions with special meanings when linked with debug library (Continued)

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

Note: For your final release build, you must implement the functionality of the functions used by your application.

THE C-SPY TERMINAL I/O WINDOW

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

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

See the IAR Embedded Workbench® IDE for MSP430 User Guide for more information about the Terminal I/O window.

Speeding up terminal output

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

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

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

Overriding library modules
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 430\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:
   ```
   icc430 library_module.c
   ```
   This creates a replacement object module file named library_module.r43.

   **Note:** Make sure to use a library that matches the settings of the rest of your application.

4. Add library_module.r43 to the XLINK command line, either directly or by using an extended linker command file, for example:
   ```
   xlink library_module.r43 dl430fn.r43
   ```
   Make sure that library_module.r43 is placed before the library on the command line. This ensures that your module is used instead of the one in the library.
Building and using a customized library

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

- You want to build a library using different optimization settings, hardware multiplier support, or register locking settings than the prebuilt libraries
- You want to define your own library configuration with support for locale, file descriptors, multibyte characters, et cetera.

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).

For information about the build process and the IAR Command Line Build Utility, see the IAR Embedded Workbench® IDE for MSP430 User Guide.

SETTING UP A LIBRARY PROJECT

The IDE provides library project templates for all prebuilt libraries. Note that when you create a new library project from a template, the majority of the files included in the new project are the original installation files. If you are going to modify these files, make copies of them first and replace the original files in the project with these copies.

In the IDE, modify the generic options in the created library project to suit your application, see Basic 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 libraryname.h, which sets up that specific library with full library configuration. For more information, see Table 10, Customizable items, page 49.

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

Modifying the library configuration file

In your library project, open the file libraryname.h and customize it by setting the values of the configuration symbols according to the application requirements.

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

USING A CUSTOMIZED LIBRARY

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

In the IDE you must do these steps:

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

System startup and termination

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

The code for handling startup and termination is located in the source files cstartup.s43 and low_level_init.c located in the 430\src\lib directory.

For information about how to customize the system startup code, see Customizing system initialization, page 61.
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 pointer (`SP`) is initialized.
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations. (Make sure that the MSP430 Watchdog mechanism is disabled during initialization.)
For the C/C++ initialization, it looks like this:

- The RAM area of __ramfunc functions is initialized
- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if __low_level_init returns zero. For more details, see *Initialized data*, page 37
- 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:

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

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

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

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

An application can also exit by calling the `abort` or the `_Exit` function. The `abort` function just calls `__exit` to halt the system, and does not perform any type of cleanup. The `_Exit` function is equivalent to the `abort` function, except for the fact that `_Exit` takes an argument for passing exit status information.
If you want your application to do anything extra at exit, for example resetting the system, you can write your own implementation of the `__exit(int)` function.

**C-SPY interface to system termination**

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *Application debug support*, page 52.

### 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.s43` 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.s43` and `low_level_init.c`, located in the `430\src\lib` directory.

**Note:** Normally, you do not need to customize either of the files `cstartup.s43` or `cexit.s43`.

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 56.

**Note:** Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s43`, 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`. Note that static initialized variables cannot be used within the file, because variable initialization has not been performed at this point.

The value returned by `__low_level_init` determines whether or not data segments should be initialized by the system startup code. If the function returns 0, the data segments will not be initialized.

**Note:** The file `intrinsics.h` must be included by `low_level_init.c` to assure correct behavior of the `__low_level_init` routine.
MODIFYING THE FILE CSTARTUP.S43

As noted earlier, you should not modify the file cstartup.s43 if a customized version of __low_level_init is enough for your needs. However, if you do need to modify the file cstartup.s43, 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 54.

Note that you must make sure that the linker uses the start label used in your version of cstartup.s43. For information about how to change the start label used by the linker, read about the -s option in the IAR Linker and Library Tools Reference Guide.

Library configurations

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

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

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

These predefined library configurations are available:

<table>
<thead>
<tr>
<th>Library configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal DLIB (default)</td>
<td>No locale interface, C locale, no file descriptor support, 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 15: Library configurations

CHOOSING A RUNTIME CONFIGURATION

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

- Default prebuilt configuration—if you do not specify a library configuration explicitly you will get the default configuration. A configuration file that matches the runtime library object file will automatically be used.
● Prebuilt configuration of your choice—to specify a runtime configuration explicitly, use the --dlib_config compiler option. See --dlib_config, page 165.

● Your own configuration—you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see Building and using a customized library, page 56.

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

### Standard streams for input and output

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

There are 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 430\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 56. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

**Note:** If you write your own variants of __read or __write, special considerations for the C-SPY runtime interface are needed, see Application debug support, page 52.

#### 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 @ 0xD2;
size_t __write(int handle, const unsigned char *buf,
```
size_t __read(int handle, size_t bufSize)
{
    size_t nChars = 0;
    /* Check for the command to flush all handles */
    if (handle == -1)
    {
        return 0;
    }
    /* Check for stdout and stderr
      * (only necessary if FILE descriptors are enabled.) */
    if (handle != 1 && handle != 2)
    {
        return -1;
    }
    for (;; /* Empty */; bufSize > 0; --bufSize)
    {
        lcdIO = *buf;
        ++buf;
        ++nChars;
    }
    return nChars;
}

Note: A call to __write where buf has the value NULL is a command to flush the handle. When the handle is -1, all streams should be flushed.

Example of using __read

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

#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0xD2;

size_t __read(int handle, size_t bufSize)
{
    size_t nChars = 0;
    /* Check for stdin
      * (only necessary if FILE descriptors are enabled) */
    if (handle != 0)
    {
        return -1;
    }
    /* Check for stdout and stderr
      * (only necessary if FILE descriptors are enabled.) */
    if (handle != 1 && handle != 2)
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 131.

**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 50.

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>Print configuration symbols</th>
<th>Includes support for</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DLIB_PRINTF_MULTIBYTE</td>
<td>Multibyte characters</td>
</tr>
<tr>
<td>_DLIB_PRINTF_LONG_LONG</td>
<td>Long long (ll qualifier)</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_FLOAT</td>
<td>Floating-point numbers</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_A</td>
<td>Hexadecimal floating-point numbers</td>
</tr>
<tr>
<td>_DLIB_PRINTF_SPECIFIER_N</td>
<td>Output count (%n)</td>
</tr>
<tr>
<td>_DLIB_PRINTF_QUALIFIERS</td>
<td>Qualifiers h, l, v, t, and z</td>
</tr>
<tr>
<td>_DLIB_PRINTF_FLAGS</td>
<td>Flags -, +, #, and 0</td>
</tr>
</tbody>
</table>

Table 16: Descriptions of printf configuration symbols
When you build a library, these configurations determine what capabilities the function `scanf` should have:

1. Set up a library project, see Building and using a customized library, page 56.
2. Define the configuration symbols according to your application requirements.

### File input and output

The library contains a large number of powerful functions for file I/O operations. 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 62. 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 18: 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 Application debug support, page 52.

Locale

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

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

The DLIB library can be used in two main modes:

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

locale SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries support the C locale only
All libraries with *full library configuration* have support for the locale interface. For prebuilt libraries with locale interface, it is by default only supported to switch multibyte character encoding at runtime.

Libraries with *normal library configuration* do not have support for the locale interface.

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

**CUSTOMIZING THE LOCALE SUPPORT**

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

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

**Locale configuration symbols**

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

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

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 56.

**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:

\texttt{LANG\_REGION}

or

\texttt{LANG\_REGION.ENCODING}

The \texttt{LANG} part specifies the language code, and the \texttt{REGION} part specifies a region
qualifier, and \texttt{ENCODING} specifies the multibyte character encoding that should be used.

The \texttt{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:

\begin{verbatim}
setlocale (LC_ALL, "sv_FI.UTF8");
\end{verbatim}

Environment interaction

According to the C standard, your application can interact with the environment using
the functions \texttt{getenv} and \texttt{system}.

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

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `430\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 54.

THE SYSTEM FUNCTION

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 56.

**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 *Application debug support*, page 52.

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 `430\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 54.
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 56.

### 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 `430\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 54.

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 56.

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

**Note:** If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For further information, see *Application debug support*, page 52.

### Strtod

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

1. Enable the configuration symbol `__DLIB_STRTOD_HEX_FLOAT` in the library configuration file.
2. Rebuild the library, see *Building and using a customized library*, page 56.

### Pow

The DLIB runtime library contains an alternative power function with extended precision, `powXp`. The lower-precision `pow` function is used by default. To use the `powXp` function instead, link your application with the linker option `-epowXp=pow`. 

---

Part 1. Using the compiler 71
**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 `430\src\lib` directory. For further information, see *Building and using a customized library*, page 56. To turn off assertions, you must define the symbol `NDEBUG`.

In the IDE, this symbol `NDEBUG` is by default defined in a Release project and not defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See `NDEBUG`, page 244.

**Heaps**

The runtime environment supports heaps in these memory types:

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Segment name</th>
<th>Memory attribute</th>
<th>Used by default in data model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data16</td>
<td>DATA16_HEAP</td>
<td>__data16</td>
<td>Small and Medium</td>
</tr>
<tr>
<td>Data20</td>
<td>DATA20_HEAP</td>
<td>__data20</td>
<td>Large</td>
</tr>
</tbody>
</table>

*Table 19: Heaps and memory types*

To use a specific heap, the prefix in the table is the memory attribute to use in front of `malloc`, `free`, `calloc`, and `realloc`, for example `__data16_malloc`. The default functions will use one of the specific heap variants, depending on project settings such as data model. For information about how to use a specific heap in C++, see *New and Delete operators*, page 119.

See *The heap*, page 39 for information about how to set the size for each heap.

**Hardware multiplier support**

Some MSP430 devices contain a hardware multiplier. The compiler supports this unit by means of dedicated runtime library modules.

To make the compiler take advantage of the hardware multiplier unit, choose *Project*->*Options*->*General Options*->*Target* and select a device that contains a hardware multiplier unit from the *Device* drop-down menu. Make sure that the option *Hardware multiplier* is selected.

Specify which runtime library object file to use on the XLINK command line.
In addition to the runtime library object file, you must extend the XLINK command line with an additional linker configuration file if you want support for the hardware multiplier.

To use the hardware multiplier, use the command line option -f to extend the XLINK command line with the appropriate extended command line file:

```
Note: Interrupts are disabled during a hardware-multiply operation.
```

<table>
<thead>
<tr>
<th>Linker configuration file</th>
<th>Type of hardware multiplier</th>
<th>Location in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplier.xcl</td>
<td>16 or 16s</td>
<td>0x130</td>
</tr>
<tr>
<td>multiplier32.xcl</td>
<td>32</td>
<td>0x130</td>
</tr>
<tr>
<td>multiplier32_loc2.xcl</td>
<td>32</td>
<td>0x4C0</td>
</tr>
</tbody>
</table>

Table 20: Additional linker configuration files for the hardware multiplier

**Checking module consistency**

This section introduces the concept of runtime model attributes, a mechanism used by the IAR compiler, assembler, and linker to ensure that modules are built using compatible settings.

When developing an application, it is important to ensure that incompatible modules are not used together. For example, 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 this table, the object files could (but do not have to) define the two runtime attributes `color` and `taste`:

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

Table 21: Example of runtime model attributes

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

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

USING RUNTIME MODEL ATTRIBUTES

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

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

For detailed syntax information, see `rtmodel`, page 227.

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

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

For detailed syntax information, see the MSP430 IAR Assembler Reference Guide.

Note: The predefined runtime attributes all start with two underscores. Any attribute names you specify yourself should not contain two initial underscores in the name, to eliminate any risk that they will conflict with future IAR 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>__core</td>
<td>430 or 430X</td>
<td>Corresponds to the --core option.</td>
</tr>
<tr>
<td>__data_model</td>
<td>small, medium, or large</td>
<td>Corresponds to the data model used in the project; only available for MSP430X.</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>__pic</td>
<td>yes or no</td>
<td>yes if the --pic option has been specified, no if it has not been specified.</td>
</tr>
<tr>
<td>__reg_r4</td>
<td>free or undefined</td>
<td>Corresponds to the use of the register, or undefined when the register is not used. A routine that assumes that the register is locked should set the attribute to a value other than free.</td>
</tr>
<tr>
<td>__reg_r5</td>
<td>free or undefined</td>
<td></td>
</tr>
<tr>
<td>__rt_version</td>
<td>3</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>__SystemLibrary</td>
<td>CLib or DLib</td>
<td>Identifies the runtime library that you are using.</td>
</tr>
</tbody>
</table>

Note: The value free should be seen as the opposite of locked, that is, the register is free to be used by the compiler.

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 MSP430 IAR Assembler Reference Guide.

Examples

For an example of using the runtime model attribute __rt_version for checking the module consistency as regards the used calling convention, see Hints for a quick introduction to the calling conventions, page 92.
The following assembler source code provides a function that increases the register R4 to count the number of times it was called. The routine assumes that the application does not use R4 for anything else, that is, the register is locked for usage. To ensure this, a runtime module attribute, __reg_r4, is defined with a value counter. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not set this attribute. Note that the compiler sets this attribute to free, unless the register is locked.

```assembly
rtmodel   "__reg_r4", "counter"
module    myCounter
public    myCounter
rseg      CODE:CODE:NOROOT(1)
myCounter: inc             R4
            ret
endmod
end
```

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

```
Error[e117]: Incompatible runtime models. Module myCounter specifies that '__reg_r4' must be 'counter', but module part1 has the value 'free'
```

**USER-DEFINED RUNTIME MODEL ATTRIBUTES**

In cases where the predefined runtime model attributes are not sufficient, you can use the RTMODEL assembler directive to define your own attributes. For each property, select a key and a set of values that describe the states of the property that are incompatible. Note that key names that start with two underscores are reserved by the compiler.

For example, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example uart. For each mode, specify a value, for example model 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"
```
The CLIB runtime environment

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

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

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

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

For information about migrating from CLIB to DLIB, see the IAR Embedded Workbench® Migration Guide for MSP430.

Runtime environment

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

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

- Core
- Data model
- Size of the double floating-point type
Position-independent code.

These prebuilt libraries are available:

<table>
<thead>
<tr>
<th>Library object file</th>
<th>Processor core</th>
<th>Data model</th>
<th>Size of double</th>
<th>PIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl430f.r43</td>
<td>430</td>
<td>—</td>
<td>32</td>
<td>No</td>
</tr>
<tr>
<td>cl430fp.r43</td>
<td>430</td>
<td>—</td>
<td>32</td>
<td>Yes</td>
</tr>
<tr>
<td>cl430d.r43</td>
<td>430</td>
<td>—</td>
<td>64</td>
<td>No</td>
</tr>
<tr>
<td>cl430dp.r43</td>
<td>430</td>
<td>—</td>
<td>64</td>
<td>Yes</td>
</tr>
<tr>
<td>cl430xsf.r43</td>
<td>430X</td>
<td>Small</td>
<td>32</td>
<td>No</td>
</tr>
<tr>
<td>cl430xsm.r43</td>
<td>430X</td>
<td>Small</td>
<td>64</td>
<td>No</td>
</tr>
<tr>
<td>cl430xmf.r43</td>
<td>430X</td>
<td>Medium</td>
<td>32</td>
<td>No</td>
</tr>
<tr>
<td>cl430xmd.r43</td>
<td>430X</td>
<td>Medium</td>
<td>64</td>
<td>No</td>
</tr>
<tr>
<td>cl430xlf.r43</td>
<td>430X</td>
<td>Large</td>
<td>32</td>
<td>No</td>
</tr>
<tr>
<td>cl430xld.r43</td>
<td>430X</td>
<td>Large</td>
<td>64</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 23: CLIB runtime libraries

The runtime library names are constructed in this way:

\{(lib}\{core}\{data_model}\{size_of_double}\{pic\}.r43

where

- lib is cl for the IAR CLIB runtime environment
- core is either 430 or 430x
- data_model is empty for MSP430 devices. For MSP430X devices it is one of s, m, or l, for the Small, Medium, and Large data model, respectively
- size_of_double is either f for 32 bits or d for 64 bits
- pic is either empty for no support for position-independent code or p for position-independent code.

The IDE includes the correct runtime library based on the options you select. See the IAR Embedded Workbench® IDE for MSP430 User Guide for additional information.

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

c1430d.r43
Input and output

You can customize:

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

**CHARACTER-BASED I/O**

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

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

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

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

```c
__no_init volatile unsigned char devIO @ 8;

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

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

For information about how to include your own modified version of putchar and getchar in your project build process, see Overriding library modules, page 54.

**FORMATTERS USED BY PRINTF AND SPRINTF**

The printf and sprintf functions use a common formatter, called _formatted_write. The full version of _formatted_write is very large, and provides facilities not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided in the Standard C library.
Input and output

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

FLOATS? wrong formatter installed!

_medium_write is considerably smaller than _formatted_write.

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

Specifying the printf formatter in the IDE
1 Choose Project>Options and select the General Options category. Click the Library options tab.
2 Select the appropriate Printf formatter option, which can be either Small, Medium, or Large.

Specifying the printf formatter from the command line
To use the _small_write or _medium_write formatter, add the corresponding line in the linker configuration file:

-e_small_write=_formatted_write
or
-e_medium_write=_formatted_write

To use the full version, remove the line.

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

For such applications, a much reduced version of the printf function (without sprintf) is supplied in source form in the file intwri.c. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see Overriding library modules, page 54.
FORMATTERS USED BY SCANF AND SSCANF

Similar to the printf and sprintf functions, scanf and sscanf use a common formatter, called _formatted_read. The full version of _formatted_read is very large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, an alternative smaller version is also provided.

_medium_read

The _medium_read formatter has the same functions as the full version, except that floating-point numbers are not supported. _medium_read is considerably smaller than the full version.

Specifying the scanf formatter in the IDE

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

Specifying the read formatter from the command line

To use the _medium_read formatter, add this line in the linker configuration file:

-e_medium_read=_formatted_read

To use the full version, remove the line.

System startup and termination

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

The code for handling startup and termination is located in the source files csstartup.s43 and low_level_init.c located in the 430\src\lib directory.

Note: Normally, you do not need to customize either of the files csstartup.s43 or cexit.s43.

SYSTEM STARTUP

When an application is initialized, several steps are performed:

- The stack pointer (SP) is initialized
- The custom function __low_level_init is called if you have defined it, giving the application a chance to perform early initializations
Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see Overriding library modules, page 54, in the chapter The DLIB runtime environment.

Customizing system initialization

For information about how to customize system initialization, see Customizing system initialization, page 61.

C-SPY runtime interface

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

When code at the labels \texttt{C\_PUTCHAR} and \texttt{C\_GETCHAR} is executed, data will be sent to or read from the debugger window.

For the \texttt{C\_PUTCHAR} routine, one character is taken from the output stream and written. If everything goes well, the character itself is returned, otherwise \texttt{-1} is returned.

When the label \texttt{C\_GETCHAR} is reached, C-SPY returns the next character in the input field. If no input is given, C-SPY waits until the user types some input and presses the Return key.

To make the Terminal I/O window available, the application must be linked with the XLINK option With I/O emulation modules selected. See the IAR Embedded Workbench® IDE for MSP430 User Guide.

TERMINATION

The debugger stops executing when it reaches the special label \texttt{C\_EXIT}.

---

**Hardware multiplier support**

Some MSP430 devices contain a hardware multiplier. For information about how the compiler supports this unit, see Hardware multiplier support, page 72.

**Checking module consistency**

For information about how to check module consistency, see Checking module consistency, page 73.
Checking module consistency
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 MSP430 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 for the different cores 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 MSP430 provides several ways to access low-level resources:

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

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

INTRINSIC FUNCTIONS

The compiler provides a few predefined functions that allow direct access to low-level processor operations without having to use the assembler language. These functions are known as intrinsic functions. They can be very useful in, for example, time-critical routines.
An intrinsic function looks like a normal function call, but it is really a built-in function that the compiler recognizes. The intrinsic functions compile into inline code, either as a single instruction, or as a short sequence of instructions.

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

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

**MIXING C AND ASSEMBLER MODULES**

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

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

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

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 88. The following two are covered in the section *Calling convention*, page 91.
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 100.

The recommended method for mixing C or C++ and assembler modules is described in Calling assembler routines from C, page 88, and Calling assembler routines from C++, page 90, respectively.

**INLINE ASSEMBLER**

It is possible to insert assembler code directly into a C or C++ function. The asm and __asm keywords both insert 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)
{
    while (!sFlag)
    {
        asm("MOV &PIND,&sFlag");
    }
}
```

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
Calling assembler routines from C

Auto variables cannot be accessed.

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

Calling assembler routines from C

An assembler routine that will be called from C must:

- Conform to the calling convention
- Have a `PUBLIC` entry-point label
- Be declared as external before any call, to allow type checking and optional promotion of parameters, as in these examples:
  
  ```
  extern int foo(void);
  or
  extern int foo(int i, int j);
  ```

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

CREATING SKELETON CODE

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

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

```c
extern int gInt;
extern char gChar;

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

```c
int main()
```
Assuming the code is part of a larger document, the following is a representation of the text:

```c
{  
  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:

```
icc430 skeleton.c -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `. (period)` specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s43`. Also remember to specify the data model you are using (if applicable), a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s43`.

**Note:** The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file. In the IDE, choose **Project>Options>C/C++ Compiler> List** and deselect the suboption **Include call frame information**. On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.

**The output file**

The output file contains the following important information:

- The calling convention
- The return values
- The global variables
- The function parameters
Calling assembler routines from C++

- 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 100.

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:

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

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

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

```cpp
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.

The compiler provides two calling conventions—Version1 and Version2. This section describes the calling conventions used by the compiler. These items are examined:

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

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

The compiler supports two calling conventions:

- The Version1 calling convention is used by version 1.x, 2.x, and 3.x of the compiler.
- The Version2 calling convention was introduced with version 4.x of the compiler. It is more efficient than the Version1 calling convention.

You can explicitly specify the calling convention when you declare and define functions. However, normally this is not needed, unless the function is written in assembler.

For old routines written in assembler and that use the Version1 calling convention, the function attribute `__cc_version1` should be used, for example:

```c
extern __cc_version1 void doit(int arg);
```

New routines written in assembler should be declared using the function attribute `__cc_version2`. This ensures that they will be called using the Version2 calling convention if new calling conventions are introduced in future compilers.

Hints for a quick introduction to the calling conventions

Both calling conventions are complex and if you intend to use any of them for your assembler routines, you should create a list file and see how the compiler assigns the different parameters to the available registers. For an example, see Creating skeleton code, page 88.

If you intend to use both of the calling conventions, you should also specify a value to the runtime model attribute `__rt_version` using the RTMODEL assembler directive:

```asm
RTMODEL "__rt_version", "value"
```

The parameter value should have the same value as the one used internally by the compiler. For information about what value to use, see the generated list file. If the calling convention changes in future compiler versions, the runtime model value used internally by the compiler will also change. Using this method gives a module consistency check, because the linker produces errors for mismatches between the values.

For more information about checking module consistency, see Checking module consistency, page 73.

FUNCTION DECLARATIONS

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

```c
int MyFunction(int first, char * second);
```
This means that the function takes two parameters: an integer and a pointer to a character. The function returns a value, an integer.

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

**USING C LINKAGE IN C++ SOURCE CODE**

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

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

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

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

```c
#ifdef __cplusplus
extern "C"
{
#endif
int F(int);
#ifdef __cplusplus
}
#endif
```

**PRESERVED VERSUS SCRATCH REGISTERS**

The general MSP430 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 R12 to R15, as well as the return address registers, are considered scratch registers and can be used by the function.

When the registers R11:R10:R9:R8 are used for passing a 64-bit scalar parameter, they are also considered to be scratch registers.
Preserved registers

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

The registers R4 to R11 are preserved registers.

If the registers R11:R10:R9:R8 are used for passing a 64-bit scalar parameter, they do not have to be preserved.

Note:

- When compiling for the MSP430X architecture in the Small data model, only the lower 16 bits of the registers are preserved, unless the __save_reg20 attribute is specified. It is only necessary to save and restore the upper 4 bits of you have an assembler routine that uses these bits.
- When compiling using the options --lock_r4 or --lock_r5, the R4 and R5 registers are not used.

The stack pointer register

You must consider that the stack pointer register must at all times point to the last element on the stack or below. In the eventuality of an interrupt, which can occur at any time, 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 as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. The parameters are also passed on the stack in these cases:

- Structure types: struct, union, and classes
- Unnamed parameters to variable length (variadic) functions; in other words, functions declared as foo(param1, ...), for example printf.

Note: Interrupt functions cannot take any parameters.

Hidden parameters

In addition to the parameters visible in a function declaration and definition, there can be hidden parameters:

- If the function returns a structure, the memory location where to store the structure is passed in the register R12 as a hidden parameter.
If the function is a non-static C++ member function, then the this pointer is passed as the first parameter (but placed after the return structure pointer, if there is one). The reason for the requirement that the member function must be non-static is that static member methods do not have a this pointer.

Register parameters

These registers are available for passing parameters:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Passed in registers, Version1</th>
<th>Passed in registers, Version2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit values</td>
<td>R12, R14</td>
<td>R12 to R15</td>
</tr>
<tr>
<td>16-bit values</td>
<td>R12, R14</td>
<td>R12 to R15</td>
</tr>
<tr>
<td>20-bit values</td>
<td>R12, R14</td>
<td>R12 to R15</td>
</tr>
<tr>
<td>32-bit values</td>
<td>(R13:R12), (R15:R14)</td>
<td>(R13:R12), (R15:R14)</td>
</tr>
</tbody>
</table>

Table 24: Registers used for passing parameters

Note: When you compile for the MSP430X architecture which supports 20-bit registers, it is assumed that the upper four bits of all parameter registers are zero (0), with exception for registers used for passing 8-bit and 20-bit values.

The assignment of registers to parameters is a straightforward process.

For Version1, the first parameter is assigned to R12 or R13:R12, depending on the size of the parameter. The second parameter is passed in R14 or R15:R14. Should there be no more available registers, the parameter is passed on the stack.

For Version2, each parameter is assigned to the first register that has not already been assigned a parameter, starting from R12.

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 even location on the stack. It is
the responsibility of the caller to remove the parameters from the stack by restoring the stack pointer.

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

**Figure 4: Stack image after the function call**

**Note:** The number of bytes reserved for the return address depends on the --core option, see *Calling functions*, page 99.

### 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.

**Note:** An interrupt function must have the return type `void`.

### Registers used for returning values

These registers are available for returning values:

<table>
<thead>
<tr>
<th>Return values</th>
<th>Passed in registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit values</td>
<td>R12</td>
</tr>
<tr>
<td>16-bit values</td>
<td>R12</td>
</tr>
<tr>
<td>20-bit values</td>
<td>R12</td>
</tr>
<tr>
<td>32-bit values</td>
<td>R13:R12</td>
</tr>
<tr>
<td>64-bit values</td>
<td>R15:R14:R13:R12</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

When finished, a function written in assembler language returns to the caller. At a function call, the return address is stored on the stack.

A function returns by performing any of the following instructions:

- RET instruction, when compiling for the MSP430 architecture
- RETA, when compiling for the MSP430X architecture
- RETI, for interrupt functions regardless of which architecture is being used.

If the --pic compiler option has been specified, this code sequence is used to return a function:

```
add.w   #4, 0(SP)
ret
```

Restrictions for special function types

For information about how the __interrupt, __raw, and __save_reg20 keywords affect the calling convention, see Interrupt functions, page 23.

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 R12, and the return value is passed back to its caller in the register R12.

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

```
add.w   #1,R12
reta    ; For the MSP430X architecture
```
Example 2
This example shows how structures are passed on the stack. Assume these declarations:

```c
struct MyStruct
{
    int mA;
};

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

The calling function must reserve 4 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register R12. The return value is passed back to its caller in the register R12.

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

```c
struct MyStruct
{
    int mA;
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value—typically on the stack—and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in R12. The caller assumes that this register remains untouched. The parameter `x` is passed in R13.

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 pointer, so there is no hidden parameter. The parameter `x` is passed in R12 and the return value is also returned in R12.

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 MSP430 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 MSP430 IAR Assembler Reference Guide.

## Calling functions

Functions can be called in two fundamentally different ways—directly or via a function pointer. In this section we will discuss how both types of calls will be performed.

### ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

This section presents the assembler instructions that can be used for calling and returning from functions on the MSP430 microcontroller.

When calling an assembler module from C modules, it is important to match the calling convention.

C modules use different instructions to call an external function depending on the `--core` option they were compiled with:

- with the `--core=430` option, they use the `CALL` instruction
- with the `--core=430X` option, they use the `CALLA` instruction.

These two call instructions have different stack layouts:

- The `CALL` instruction pushes a 2-byte return address on the stack
- The `CALLA` instruction pushes a 4-byte return address on the stack.

This must be matched in your assembler routine by using the corresponding `RET` and `RETA` return instructions, or the function will not return properly which leads to a corrupt stack.

**Note:** Interrupt functions written in assembler are not affected, because all interrupt routines must return using the `RETI` instruction, regardless of which architecture that you are using.

Because the calling convention differs slightly between the two architectures, you can define the runtime attribute `_core` in all your assembler routines, to avoid inconsistency. Use one of the following lines:

```assembly
RTMODEL "__core","430"
RTMODEL "__core","430X"
```

Using this module consistency check, the linker will produce an error if there is a mismatch between the values.

For more information about checking module consistency, see Checking module consistency, page 73.
Position-independent code

If the --pic compiler option has been specified, this instruction sequence is used for calling a function:

```
push    PC
add     #LWRD(x-(x+4)),PC
```

The return sequence is described under Return address handling, page 97.

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 MSP430 IAR Assembler Reference Guide.

CFI DIRECTIVES

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

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

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

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

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

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFA</td>
<td>The call frames of the stack</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 call frame information` is selected.

   For the source code in this example, the list file looks like this:

   ```
   NAME Cfi
   
   RTMODEL "__SystemLibrary", "DLib"
   RTMODEL "__core", "430"
   RTMODEL "__double_size", "32"
   RTMODEL "__pic", "no"
   RTMODEL "__reg_r4", "free"
   RTMODEL "__reg_r5", "free"
   RTMODEL "__rt_version", "3"
   
   RSEG CSTACK:DATA:SORT:NOROOT(0)
   ```
EXTERN ?longjmp_r4
EXTERN ?longjmp_r5
EXTERN ?setjmp_r4
EXTERN ?setjmp_r5

PUBWEAK ?setjmp_save_r4
PUBWEAK ?setjmp_save_r5
PUBLIC cfiExample
FUNCTION cfiExample,021203H
ARGFRAME CSTACK, 0, STACK
LOCFRAME CSTACK, 4, STACK

CFI Names cfiNames0
CFI StackFrame CFA SP DATA
CFI Resource PC:16, SP:16, SR:16
CFI Resource R13:16, R14:16, R15:16
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 2
CFI DataAlign 2
CFI ReturnAddress PC CODE
CFI CFA SP+2
CFI PC Frame(CFA, -2)
CFI SR Undefined
CFI R4 SameValue
CFI R5 SameValue
CFI R6 SameValue
CFI R7 SameValue
CFI R8 SameValue
CFI R9 SameValue
CFI R10 SameValue
CFI R11 SameValue
CFI R12 Undefined
CFI R13 Undefined
CFI R14 Undefined
CFI R15 Undefined
CFI EndCommon cfiCommon0

EXTERN F
FUNCTION F,0202H
Assembler language interface

RSEG CODE:CODE:REORDER:NOROOT(1)
cfiExample:
  CPI Block cfiBlock0 Using cfiCommon0
  CPI Function cfiExample

  FUNCALL cfiExample, F
  LOCFRAME CSTACK, 4, STACK
  PUSH.W R10
  CPI R10 Frame(CFA, -4)
  CPI CPA SP+4
  MOV.W R12, R10

  CALL   #F
  ADD.W R10, R12
  POP.W R10
  CPI R10 SameValue
  CPI CPA SP+2
  RET
  CPI EndBlock cfiBlock0

RSEG CODE:CODE:REORDER:NOROOT(1)
?setjmp_save_r4:
  REQUIRE ?setjmp_r4
  REQUIRE ?longjmp_r4

RSEG CODE:CODE:REORDER:NOROOT(1)
?setjmp_save_r5:
  REQUIRE ?setjmp_r5
  REQUIRE ?longjmp_r5

END

Note: The header file cfi.m43 contains the macros XCFI_NAMES and XCFI_COMMON, which declare a typical names block and a typical common block. These two macros declare several resources, both concrete and virtual.
Using C

This chapter gives an overview of the compiler’s support for the C language. The chapter also gives a brief overview of the IAR C language extensions.

C language overview

The IAR C/C++ Compiler for MSP430 supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as Standard C and is the default standard used in the compiler. This standard is stricter than C89.

In addition, the compiler also supports the ISO 9899:1990 standard (including all technical corrigenda and addenda), also known as C94, C90, C89, and ANSI C. In this guide, this standard is referred to as C89. Use the --c89 compiler option to enable this standard.

The C99 standard is derived from C89, but adds features like these:

- The *inline* keyword advises the compiler that the function declared immediately after the directive should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a *for* loop
- The *bool* data type
- The *long long* data type
- The *complex* floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator *_Pragma()*
- Variadic macros, which are the preprocessor macro equivalents of *printf* style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option --vla
- Inline assembler using the *asm* or the *__asm* keyword.
Note: Even though it is a C99 feature, the IAR C/C++ Compiler for MSP430 does not support UCNs (universal character names).

**Inline assembler**

Inline assembler can be used for inserting assembler instructions in the generated function.

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

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

The syntax is:

```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"  
     " jmp Label");
```

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

For more information about inline assembler, see *Mixing C and assembler*, page 85.

---

**Extensions overview**

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

This is an overview of the available extensions:

- IAR C language extensions
  
  For a summary of available language extensions, see *IAR C language extensions*, page 108. 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 Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.
The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For a list of available pragma directives, see the chapter Pragma directives.

- **Pragma directives**
  The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler.

- **Preprocessor extensions**
  The preprocessor of the compiler adheres to Standard C. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter The preprocessor.

- **Intrinsic functions**
  The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see Mixing C and assembler, page 85. For a list of available functions, see the chapter Intrinsic functions.

- **Library functions**
  The IAR DLIB Library provides the C and C++ library definitions that apply to embedded systems. For more information, see IAR DLIB Library, page 249.

**Note:** Any use of these extensions, except for the pragma directives, makes your source code inconsistent with Standard C.

### ENABLING LANGUAGE EXTENSIONS

You can choose different levels of language conformance by means of project options:

<table>
<thead>
<tr>
<th>Command line</th>
<th>IDE*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>--strict</strong></td>
<td><strong>Strict</strong></td>
<td>All IAR C language extensions are disabled; errors are issued for anything that is not part of Standard C.</td>
</tr>
<tr>
<td>None</td>
<td><strong>Standard</strong></td>
<td>All extensions to Standard C are enabled, but no extensions for embedded systems programming. For a list of extensions, see IAR C language extensions, page 108.</td>
</tr>
<tr>
<td>-e</td>
<td><strong>Standard with IAR extensions</strong></td>
<td>All IAR C language extensions are enabled.</td>
</tr>
</tbody>
</table>

*In the IDE, choose Project>Options> C/C++ Compiler>Language>Language conformance and select the appropriate option. Note that language extensions are enabled by default.*
The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- **Extensions for embedded systems programming**—extensions specifically tailored for efficient embedded programming for the specific microcontroller you are using, typically to meet memory restrictions
- **Relaxations to Standard C**—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions, see Relaxations to Standard C, page 110.

### EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

The following language extensions are available both in the C and the C++ programming languages and they are well suited for embedded systems programming:

- **Memory attributes, type attributes, and object attributes**
  For information about the related concepts, the general syntax rules, and for reference information, see the chapter Extended keywords.

- **Placement at an absolute address or in a named segment**
  The @ operator or the directive #pragma location can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these features, see Controlling data and function placement in memory, page 131, and location, page 222.

- **Alignment control**
  Each data type has its own alignment; for more details, see Alignment, page 189. If you want to change the alignment, the #pragma pack and #pragma data_alignment directives are available. If you want to check the alignment of an object, use the __ALIGNOF__( ) operator.
  The __ALIGNOF__ operator is used for accessing the alignment of an object. It takes one of two forms:
  - __ALIGNOF__( type )
  - __ALIGNOF__( expression )
  In the second form, the expression is not evaluated.

- **Anonymous structs and unions**
  C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see Anonymous structs and unions, page 129.
• Bitfields and non-standard types

In Standard C, a bitfield must be of the type `int` or `unsigned int`. Using IAR C language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. For more information, see Bitfields, page 191.

Dedicated segment operators

The compiler supports getting the start address, end address, and size for a segment with these built-in segment operators: `__segment_begin`, `__segment_end`, and `__segment_size`.

These operators behave syntactically as if declared like:

```c
void * __segment_begin(char const * segment)
void * __segment_end(char const * segment)
size_t __segment_size(char const * segment)
```

The operators can be used on named sections defined in the linker file.

The `__segment_begin` operator returns the address of the first byte of the named segment.

The `__segment_end` operator returns the address of the first byte after the named segment.

The `__segment_size` operator returns the size of the named segment in bytes.

When you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment in the linker configuration file, the segment operators can be used for getting the start and end address of the memory range where the segments were placed.

The named `segment` must be a string literal and it must have been declared earlier with the `#pragma segment` directive. If the segment was declared with a memory attribute `memattr`, the type of the `__segment_begin` operator is a pointer to `memattr void`. Otherwise, the type is a default pointer to `void`. Note that you must enable language extensions to use these operators.

Example

In this example, the type of the `__segment_begin` operator is `void __data16 *`:

```c
#pragma segment="MYSEGMENT" __data16
...
segment_start_address = __segment_begin("MYSEGMENT");
```

See also `segment`, page 228, and `location`, page 222.
RELAXATIONS TO STANDARD C

This section lists and briefly describes the relaxation of some Standard C issues and also some useful but minor syntax extensions:

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

- **Forward declaration of `enum` types**
  The extensions allow you to first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- **Accepting missing semicolon at the end of a `struct` or `union` specifier**
  A warning—instead of an error—is issued if the semicolon at the end of a `struct` or `union` specifier is missing.

- **Null and `void`**
  In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In Standard C, some operators allow this kind of behavior, while others do not allow it.

- **Casting pointers to integers in static initializers**
  In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see Casting, page 195.

- **Taking the address of a register variable**
  In Standard C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.

- **`long float` means `double`**
  The type `long float` is accepted as a synonym for `double`.

- **Repeated `typedef` declarations**
  Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- **Mixing pointer types**
  Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.
• Non-top level `const`
  Assignment of pointers is allowed in cases where the destination type has added type
  qualifiers that are not at the top level (for example, `int **` to `int const **`).
  Comparing and taking the difference of such pointers is also allowed.

• Non-lvalue arrays
  A non-lvalue array expression is converted to a pointer to the first element of the
  array when it is used.

• Comments at the end of preprocessor directives
  This extension, which makes it legal to place text after preprocessor directives, is
  enabled unless the strict Standard C mode is used. The purpose of this language
  extension is to support compilation of legacy code; we do not recommend that you
  write new code in this fashion.

• An extra comma at the end of `enum` lists
  Placing an extra comma is allowed at the end of an `enum` list. In strict Standard C
  mode, a warning is issued.

• A label preceding a `}`
  In Standard C, a label must be followed by at least one statement. Therefore, it is
  illegal to place the label at the end of a block. The compiler allows this, but issues a
  warning.
  Note that this also applies to the labels of `switch` statements.

• Empty declarations
  An empty declaration (a semicolon by itself) is allowed, but a remark is issued
  (provided that remarks are enabled).

• Single-value initialization
  Standard C requires that all initializer expressions of static arrays, structs, and unions
  are enclosed in braces.
  Single-value initializers are allowed to appear without braces, but a warning is
  issued. The compiler accepts this expression:

```
struct str
{
  int a;
} x = 10;
```
● Declarations in other scopes
External and static declarations in other scopes are visible. In the following example, the variable \( y \) can be used at the end of the function, even though it should only be visible in the body of the \texttt{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 \texttt{__func__} or \texttt{__FUNCTION__} inside a function body to make the symbol expand into a string, with the function name as context. Use the symbol \texttt{__PRETTY_FUNCTION__} to also include the parameter types and return type. The result might, for example, look like this if you use the \texttt{__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 \texttt{-e}, page 167.

● Static functions in function and block scopes
Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

● Numbers scanned according to the syntax for numbers
Numbers are scanned according to the syntax for numbers rather than the \texttt{pp-number} syntax. Thus, \texttt{0x123e+1} is scanned as three tokens instead of one valid token. (If the \texttt{--strict} option is used, the \texttt{pp-number} syntax is used instead.)
Using C++

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

Overview

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

STANDARD EMBEDDED C++

The following C++ features are supported:

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

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

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

- Templates
- Multiple and virtual inheritance
- Exception handling
Overview

IAR C/C++ Compiler Reference Guide for MSP430

- 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`, `std::exception`, and `typeinfo`) are excluded.

Note: The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

EXTENDED EMBEDDED C++

IAR Systems’ Extended EC++ is a slightly larger subset of C++ which adds these features to the standard EC++:

- Full template support
- Namespace support
- The `mutable` attribute
- The cast operators `static_cast`, `const_cast`, and `reinterpret_cast`.

All these added features conform to the C++ standard.

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++ language, which means no exceptions, no multiple inheritance, and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

Note: A module compiled with Extended EC++ enabled is fully link-compatible with a module compiled without Extended EC++ enabled.

ENABLING C++ SUPPORT

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

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

Feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for MSP430, 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 199.

**Example**

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

    // A static task function
    static __task void F();

    // A task function
    __task void G();
}
The this pointer

The this pointer used for referring to a class object or calling a member function of a class object will by default have the data memory attribute for the default data pointer type. This means that such a class object can only be defined to reside in memory from which pointers can be implicitly converted to a default data pointer. This restriction might also apply to objects residing on a stack, for example temporary objects and auto objects.

Class memory

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

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

Example

class __data20 C
{
  public:
    void MyF(); // Has a this pointer of type C __data20 *
    void MyF() const; // Has a this pointer of type C __data20 const *
    C(); // Has a this pointer pointing into data20 memory
    C(C const &); // Takes a parameter of type C __data20 const & (also true of generated copy constructor)
    int mI;
};
C Ca;               // Resides in data20 memory instead of the
// default memory
C __data16 Cb;     // Resides in data16 memory, the 'this'
// pointer still points into data20 memory

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

C *Cp1;             // Creates a pointer to data20 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:

class __data20 C;

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

A built-in operator returns the class memory type associated with a class,
__memory_of(class). For instance, __memory_of(C) returns __data20.

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

class __data20 D : public C
{ // OK, same class memory
public:
    void MyG();
    int mJ;
};

class __data16 E : public C
{ // OK, data16 memory is inside data20
public:
    void MyG() // Has a this pointer pointing into data16 memory
    { MyF(); // Gets a this pointer into data20 memory
    int mJ;
    };

class F : public C
{ // OK, will be associated with same class memory as C
public:
    void MyG();
};
int mJ;
};

Note that the following is not allowed because data20 is not inside data16 memory:

class __data20 G:public C
{
};

A new expression on the class will allocate memory in the heap associated with the class memory. A delete expression will naturally deallocate the memory back to the same heap. To override the default new and delete operator for a class, declare

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

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

For more information about memory types, see Memory types (MSP430X only), page 15.

FUNCTION TYPES

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

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

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

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

void MyG()
{
    MyF(F1); // Always works
    MyF(F2); // FpCpp is compatible with FpC
}
NEW AND DELETE OPERATORS

There are operators for `new` and `delete` for each memory that can have a heap, that is, `data16` and `data20` memory.

```cpp
// Assumes that there is a heap in both data16 and data20 memory
void __data20 *operator new __data20(__data20_size_t);
void __data16  *operator new __data16 (__data16_size_t);
void operator delete(void __data20 *);
void operator delete(void __data16  *);

// And correspondingly for array new and delete operators
void __data20 *operator new[] __data20(__data20_size_t);
void __data16  *operator new[] __data16 (__data16_size_t);
void operator delete[](void __data20 *);
void operator delete[](void __data16  *);
```

Use this syntax if you want to override both global and class-specific `operator new` and `operator delete` for any data memory.

Note that there is a special syntax to name the `operator new` functions for each memory, while the naming for the `operator delete` functions relies on normal overloading.

New and delete expressions

A `new` expression calls the `operator new` function for the memory of the type given. If a class, struct, or union type with a class memory is used, the class memory will determine the `operator new` function called. For example,

```cpp
void MyF()
{
  // Calls operator new __data16(__data16_size_t)
  int __data16 *p = new __data16 int;

  // Calls operator new __data16(__data16_size_t)
  int __data16 *q = new int __data16;

  // Calls operator new[] __data16(__data16_size_t)
  int __data16 *r = new __data16 int[10];

  // Calls operator new __data20(__data20_size_t)
  class __data20 S
  {
  }
  S *s = new S;

  // Calls operator delete(void __data16 *)
  delete p;
}
```
// Calls operator delete(void __data20 *)
delete s;

int __data20 *t = new __data16 int;
delete t; // Error: Causes a corrupt heap
}

Note that the pointer used in a delete expression must have the correct type, that is, the same type as that returned by the new expression. If you use a pointer to the wrong memory, the result might be a corrupt heap.

**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.

**Templates and data memory attributes**

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling were changed—class template partial specialization matching and function template parameter deduction.

In Extended Embedded C++, the class template partial specialization matching algorithm works like this:

*When a pointer or reference type is matched against a pointer or reference to a template parameter type, the template parameter type will be the type pointed to, stripped of any data memory attributes, if the resulting pointer or reference type is the same.*

**Example**

// We assume that data20 is the memory type of the default pointer.
template< typename > class Z;
template< typename T > class Z<T * >;

Z<int __data16 *> zn; // T = int __data16
Z<int __data20 *> zf; // T = int
In Extended Embedded C++, the function template parameter deduction algorithm works like this:

*When function template matching is performed and an argument is used for the deduction; if that argument is a pointer to a memory that can be implicitly converted to a default pointer, do the parameter deduction as if it was a default pointer.*

*When an argument is matched against a reference, do the deduction as if the argument and the parameter were both pointers.*

**Example**

```c++
template<typename T> void fun(T *);
fun((int __data16 *) 0);  // T = int. The result is different
  // than the analogous situation with
  // class template specializations.
fun((int *) 0);    // T = int
fun((int __data20 *) 0); // T = int
```

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to small memory types. For large and “other” memory types (memory that cannot be pointed to by a default pointer) it is possible. To make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

**Example**

```c++
#pragma basic_template_matching
template<typename T> void fun(T *);
fun((int __data16 *) 0); // T = int __data16
```

**Non-type template parameters**

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

**Example**

```c++
#include <intrinsics.h>
__no_init int __regvar x @ __R4;

template<__regvar int &y>
void foo()
```
\{ 
   y = 17;
\}

void bar()
{
   foo<x>();
}

Note: This example must be compiled with the \texttt{--regvar_r4} compiler option.

\section*{The standard template library}

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in \textit{Extended Embedded C++}, page 114.

The containers in the STL, like \texttt{vector} and \texttt{map}, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

\begin{itemize}
   \item The container itself will reside in the chosen memory
   \item Allocations of elements in the container will use a heap for the chosen memory
   \item All references inside it use pointers to the chosen memory
\end{itemize}

\textbf{Example}

\begin{verbatim}
vector<int> d; // d placed in default memory, using
   // the default heap, uses default
   // pointers
vector<int __data16> __data16 x; // x placed in data16 memory,
   // heap allocation from data16, uses
   // pointers to data16 memory
vector<int __data20> __data16 y; // y placed in data16 memory,
   // heap allocation from data20, uses
   // pointers to data20 memory
vector<int __data16> __data20 z; // Illegal
\end{verbatim}

Note also that \texttt{map} with \texttt{multimap}, \texttt{hash_map}, and \texttt{hash_multimap} all use the memory of \texttt{T}. This means that the value type of these collections will be \texttt{pair<key, const T>} \texttt{mem} where \texttt{mem} is the memory type of \texttt{T}. Supplying a key with a memory type is not useful.
Example

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated assign member method must be used.

```c++
vector<int __data16> x;
vector<int __data20> y;
x = y; // Illegal
y = x; // Illegal
```

However, the templated assign member method will work:

```c++
x.assign(y.begin(), y.end());
y.assign(x.begin(), x.end());
```

STL and the IAR C-SPY® Debugger

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

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

To read more about displaying STL containers in the C-SPY debugger, see the IAR Embedded Workbench® IDE for MSP430 User Guide.

VARIANTS OF CAST OPERATORS

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

```c++
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

MUTABLE

The mutable attribute is supported in Extended EC++. A mutable symbol can be changed even though the whole class object is const.

NAMESPACE

The namespace feature is only supported in Extended EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the std namespace.
THE STD NAMESPACE

The std namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the std namespace, simply define std as nothing; for example:

#define std

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

USING INTERRUPTS AND EC++ DESTRUCTORS

If interrupts are enabled and the interrupt functions use static class objects that need to be destroyed (using destructors), there might be problems if the interrupt occur during or after application exits. If an interrupt occurs after the static class object was destroyed, the application will not work properly.

To avoid this, make sure that interrupts are disabled when returning from main or when calling exit or abort. To do this, call the intrinsic function __disable_interrupt.

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:

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

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

```cpp
class A
{
    const int mSize = 10; //Possible when using IAR language extensions
    int mArr[mSize];
};
```

According to the standard, initialized static data members should be used instead.
● In the declaration of a class member, a qualified name can be used, for example:

```cpp
struct A
{
    int A::F(); // Possible when using IAR language extensions
    int G();   // According to standard
};
```

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

```cpp
extern "C" void F(); // Function with C linkage
void (*PF)()       // PF points to a function with C++ linkage
    = &F; // Implicit conversion of function pointer.
```

According to the standard, the pointer must be explicitly converted.

● If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```cpp
bool X;
char *P1 = X ? "abc" : "def"; // Possible when using IAR language extensions
char const *P2 = X ? "abc" : "def"; // According to standard
```

● Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.

● In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.

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

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

The topics discussed are:

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

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

Selecting data types

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

**USING EFFICIENT DATA TYPES**

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

- Use small data types.
- Try to avoid 64-bit data types, such as `double` and `long long`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- Using floating-point types is very inefficient, both in terms of code size and execution speed. If possible, consider using integer operations instead.
Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

**FLOATING-POINT TYPES**

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

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

In the compiler, the floating-point type `float` always uses the 32-bit format, and the type `double` always uses the 64-bit format. The format used by the `double` floating-point type depends on the setting of the `--double` compiler option.

Unless the application requires the extra precision that 64-bit floating-point numbers give, we recommend using 32-bit floating-point numbers instead. Also consider replacing code using floating-point operations with code using integers since these are more efficient.

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

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

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

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

For more information about floating-point types, see `Floating-point types`, page 193.
ALIGNMENT OF ELEMENTS IN A STRUCTURE

The MSP430 microcontroller requires that when accessing data in memory, the data 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 189.

There are two ways to solve the problem:

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

For further details about the #pragma pack directive, see pack, page 225.

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 MSP430 they can be used in C if language extensions are enabled.

In the IDE, language extensions are enabled by default.

Use the -e compiler option to enable language extensions. See -e, page 167, for additional information.

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 IOPORT;
    struct
    {
        unsigned char Way: 1;
        unsigned char Out: 1;
    };
} @ 8;
/* Here the variables are used*/
void Test(void)
{
    IOPORT = 0;
    Way = 1;
    Out = 1;
}
```

This declares an I/O register byte `IOPORT` at address 0. The I/O register has 2 bits declared, `Way` and `Out`. Note that both the inner structure and the outer union are anonymous.

Anonymous structures and unions are implemented in terms of objects named after the first field, with a prefix `_A_` to place the name in the implementation part of the namespace. In this example, the anonymous union will be implemented through an object named `_A_IOPORT`. 
Controlling data and function placement in memory

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

- **Data models (MSP430X only)**
  Use the compiler option for selecting a data model, to take advantage of the different addressing modes available for the microcontroller and thereby also place data objects in different parts of memory. To read more about data models, see Data models (MSP430X only), page 14.

- **Memory attributes (MSP430X only)**
  Use memory attributes to override the default addressing mode and placement of individual data objects. To read more about memory attributes for data, see Using data memory attributes, page 16.

- **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.

- **The --segment option**
  Use the --segment option to place functions and/or data objects in named segments, which is useful, for example, if you want to direct them to different fast or slow memories. In contrast to using the @ operator or the #pragma location directive, there are no restrictions on what types of variables that can be placed in named segments when using the --segment option. To read more about the --segment option, see --segment, page 184.

At compile time, data and functions are placed in different segments as described in Data segments, page 35, and Code segments, page 41, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various
segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker configuration file, as described in Placing segments in memory, page 32.

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:

```
__no_init volatile char alpha @ 0x200; /* OK */
```

These examples contain two const declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0x202
__no_init const int beta; /* OK */

const int gamma @ 0x204 = 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 @ 0x206; /* Error, neither */
/* "__no_init" nor "const".*/
__no_init int epsilon @ 0x207; /* Error, misaligned. */
```

**C++ considerations**

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

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

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

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

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

**Note:** C++ static member variables can be placed at an absolute address just like any other static variable.

**DATA AND FUNCTION PLACEMENT IN SEGMENTS**

The following methods 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 `__no_init`, `const`, or `__persistent`. If declared `const`, they can have initializers.
- The `--segment` option can be used for placing variables and functions, which are parts of the whole compilation unit, in named segments.

C++ static member variables can be placed in named segments just like any other static variable.
If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker configuration file using the -Z or the -P segment control directives.

**Note:** Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

The location of the segments can be controlled from the linker configuration file.

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

**Examples of placing variables in named segments**

In the following examples, a data object is placed in a user-defined segment. The variable will be treated as if it is located in the default memory. Note that you must place the segment accordingly in the linker configuration file.

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

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

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

As usual, you can use memory attributes to direct the variable to a non-default memory (and then also place the segment accordingly in the linker configuration file):

```c
__data20 __no_init int alpha @ "MY_DATA20_NOINIT"; /* Placed in data20 */
```

This example shows incorrect usage:

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

**Examples of placing functions in named segments**

```c
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

#pragma location="MY_FUNCTIONS"
void h(void);
```
Controlling compiler optimizations

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

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

SCOPE FOR PERFORMED OPTIMIZATIONS

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

You can also exclude individual functions from the performed optimizations. The #pragma optimize directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to optimize, page 224, 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 171.

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 165.
OPTIMIZATION LEVELS

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

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (Best debug support)</td>
<td>Variables live through their entire scope</td>
</tr>
<tr>
<td></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>
<tr>
<td>Medium</td>
<td>Same as above, and:</td>
</tr>
<tr>
<td></td>
<td>Live-dead analysis and optimization</td>
</tr>
<tr>
<td></td>
<td>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>High (Balanced)</td>
<td>Same as above, and:</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 28: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see Fine-tuning enabled transformations, page 137.

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

SPEED VERSUS SIZE

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

**FINE-TUNING ENABLED TRANSFORMATIONS**

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

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

**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 174.

**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 177.

**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
Controlling compiler optimizations

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

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed. Normally, code size does not increase when optimizing for size. To control the heuristics for individual functions, use the #pragma inline directive or the Standard C inline keyword.

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 174. For information about the pragma directive, see inline, page 220.

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

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

To read more about the command line option, see --no_tbaa, page 176.
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.

### Facilitating good code generation

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

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

### WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
The compiler is capable of inlining functions, see Function inlining, page 137. To maximize the effect of the inlining transformation, it is good practice to place the definitions of small functions called from more than one module in the header file rather than in the implementation file. Alternatively, you can use multi-file compilation. For more information, see Multi-file compilation units, page 135.

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 85.

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.
- Use the --reduced_stack_space option. This will eliminate holes in the stack resulting from normal optimizations.

FUNCTION PROTOTYPES

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

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

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

To make the compiler verify that all functions have proper prototypes, use the compiler option Require prototypes (--require_prototypes).
Prototyped style
In prototyped function declarations, the type for each parameter must be specified.

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

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

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

For example:

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

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

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

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 16-bit integer, and two’s complement is assumed:

```
void F1(unsigned char c1)
{
    if (c1 == -0x80)
    ;
}
```
Here, the test is always false. On the right hand side, 0x80 is 0x0080, and ~0x0080 becomes 0xFF7F. On the left hand side, cl 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 199.

For sequences of accesses to variables that you do not want to be interrupted, use the `__monitor` keyword. This must be done for both write and read sequences, otherwise you might end up reading a partially updated variable. Accessing a small-sized `volatile` variable can be an atomic operation, but you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation. For more details, see `__monitor`, page 207.

To read more about the `volatile` type qualifier and the rules for accessing `volatile` objects, see *Declaring objects volatile*, page 199.

*Protecting the eeprom write mechanism*

A typical example of when it can be necessary to use the `__monitor` keyword is when protecting the eeprom write mechanism, which can be used from two threads (for example, main code and interrupts).

**ACCESSING SPECIAL FUNCTION REGISTERS**

Specific header files for several MSP430 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:** Header files named `mapdevice.h` are available for backward compatibility. These must be used by your assembler files.
The header file contains definitions that include bitfields, so individual bits can be accessed. The following example is from `io430x14x.h`:

```c
/* Watchdog Timer Control */
__no_init volatile union
{
    unsigned short WDTCTL;
    struct
    {
        unsigned short WDTIS0        : 1;
        unsigned short WDTIS1        : 1;
        unsigned short WDTSSSEL      : 1;
        unsigned short WDTCSNCTL     : 1;
        unsigned short WDTTMSEL      : 1;
        unsigned short WDTNMI        : 1;
        unsigned short WDTNMIES      : 1;
        unsigned short WDTHOLD       : 1;
        unsigned short               : 8;
    } WDTCTL_bit;
} @ 0x0120;

enum {
    WDTIS0    = 0x0001,
    WDTIS1    = 0x0002,
    WDTSSSEL  = 0x0004,
    WDTCSNCTL = 0x0008,
    WDTTMSEL  = 0x0010,
    WDTNMI    = 0x0020,
    WDTNMIES  = 0x0040,
    WDTHOLD   = 0x0080
};

#define WDTPW               (0x5A00)
```

By including the appropriate include file in your source code, you make it possible to access either the object or any individual bit (or bitfields) from C code as follows:

```c
/* Object access */
WDTCTL = 0x1234;

/* Bitfield accesses */
WDTCTL_bit.WDTSSEL = 1;
```

If more than one bit must be written to a memory-mapped peripheral unit at the same time, for instance to stop the watchdog timer, the defined bit constants can be used instead, for example:

```c
WDTCTL = WDTPW + WDTHOLD; /* Stop watchdog timer */
```
You can also use the header files as templates when you create new header files for other MSP430 devices. For details about the `@` operator, see *Located data*, page 41.

**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 208. Note that to use this keyword, language extensions must be enabled; see `-e`, page 167. For information about the `#pragma object_attribute`, see page 223.

**EFFICIENT SWITCH STATEMENTS**

The compiler provides a way to generate very efficient code for switch statements when it is known that the value in the expression is even and within a specific limit. This can for example be used for writing efficient interrupt service routines that use the Interrupt Vector Generators Timer A, Timer B, the I2C module, and the ADC12 module. For more information, see *Interrupt Vector Generator interrupt functions*, page 25.
Part 2. Reference information

This part of the IAR C/C++ Compiler Reference Guide for MSP430 contains these chapters:

● External interface details
● Compiler options
● Data representation
● 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 for MSP430 User Guide* for information about using the compiler from the IDE.

**COMPILER INVOCATION SYNTAX**

The invocation syntax for the compiler is:

```
icc430 [options] [sourcefile] [options]
```

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

```
icc430 prog.c --debug
```

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

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

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

**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 `icc430` command, either before or after the source filename; see *Invocation syntax*, page 147.
Include file search procedure

- Via environment variables
  The compiler automatically appends the value of the environment variables to every command line; see Environment variables, page 148.
- Via a text file, using the -f option; see -f, page 168.

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

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_INCLUDE</td>
<td>Specifies directories to search for include files; for example:</td>
</tr>
<tr>
<td></td>
<td>C_INCLUDE=c:\program files\iar systems\embedded workbench 6.n\430\inc;c:\headers</td>
</tr>
<tr>
<td>QCC430</td>
<td>Specifies command line options; for example: QCC430=-lA asm.lst</td>
</tr>
</tbody>
</table>

Table 29: Compiler environment variables

Include file search procedure

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

- If the name of the #include file is an absolute path, that file is opened.
- If the compiler encounters the name of an #include file in angle brackets, such as:
  #include <stdio.h>
  it searches these directories for the file to include:
    1 The directories specified with the -I option, in the order that they were specified, see -I, page 169.
    2 The directories specified using the C_INCLUDE environment variable, if any; see Environment variables, page 148.
    3 The automatically set up library system include directories. See --clib, page 160, --dlib, page 165, and --dlib_config, page 165.
- If the compiler encounters the name of an #include file in double quotes, for example:
  #include "vars.h"
  it searches the directory of the source file in which the #include statement occurs, and then performs the same sequence as for angle-bracketed filenames.
If there are nested `#include` files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
  #include "src.h"
  ...
src.h in directory dir\include
  #include "config.h"
  ...
```

When `dir\exe` is the current directory, use this command for compilation:
```
icc430 ..\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.

For information about the syntax for including header files, see *Overview of the preprocessor*, page 241.

---

**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 `r43`.
- Optional list files
  Various kinds of list files can be specified using the compiler option `-l`, see `-l`, page 169. By default, these files will have the filename extension `lst`.

---

External interface details

Part 2. Reference information
● Optional preprocessor output files
A preprocessor output file is produced when you use the --preprocess option; by default, the file will have the filename extension i.

● Diagnostic messages
Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. To read more about diagnostic messages, see Diagnostics, page 151.

● 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 150.

● Size information
Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as shared.
Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

### Error return codes

The compiler returns status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

<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 --warnings_affect_exit_code was used.</td>
</tr>
<tr>
<td>2</td>
<td>Errors occurred.</td>
</tr>
<tr>
<td>3</td>
<td>Fatal errors occurred, making the compiler abort.</td>
</tr>
<tr>
<td>4</td>
<td>Internal errors occurred, making the compiler abort.</td>
</tr>
</tbody>
</table>

Table 30: Error return codes
Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

MESSAGE FORMAT

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

filename,linenumber  level[tag]: message

with these elements:

- **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 construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 183.

**Warning**

A diagnostic message that is produced when the compiler 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 177.
**Error**

A diagnostic message that is produced when the compiler finds a construct which clearly violates the C or C++ language rules, such that code cannot be produced. An error will produce a non-zero exit code.

**Fatal error**

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

**SETTING THE SEVERITY LEVEL**

The diagnostic messages can be suppressed or the severity level can be changed for all diagnostics messages, except for fatal errors and some of the regular errors.

See *Summary of compiler options*, page 155, 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 for MSP430 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 147.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is optional or mandatory, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

\-O or \-Oh
For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

--misrac2004=n

**Rules for mandatory parameters**

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

-I ..\src or -I ..\src\n
For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

--diagnostics_tables=MyDiagnostics.lst

or

--diagnostics_files=MyDiagnostics.lst

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

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

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

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

-lA MyList.lst

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

--preprocess=n PreprocOutput.lst

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

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

- Options that take a filename as a parameter can optionally 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\:

  icc430 prog.c -l ..\listings\List.lst
For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name was specified with the option \texttt{-o}, in which case that name is used. For example:

\begin{verbatim}
icc430 prog.c -l ..\listings\ 
\end{verbatim}

The produced list file will have the default name \texttt{..\listings\prog.lst}

- The current directory is specified with a period (\texttt{.}). For example:

\begin{verbatim}
icc430 prog.c -l . 
\end{verbatim}

- / can be used instead of \texttt{\} as the directory delimiter.

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

\begin{verbatim}
icc430 prog.c -l - 
\end{verbatim}

### 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 \texttt{-r}:

\begin{verbatim}
icc430 prog.c -l ---r 
\end{verbatim}

- 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:

\begin{verbatim}
--diag_warning=Be0001,Be0002 
\end{verbatim}

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

\begin{verbatim}
--diag_warning=Be0001 
--diag_warning=Be0002 
\end{verbatim}

### 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>\texttt{--c89}</td>
<td>Uses the C89 standard</td>
</tr>
<tr>
<td>\texttt{--char_is_signed}</td>
<td>Treats char as signed</td>
</tr>
<tr>
<td>\texttt{--char_is_unsigned}</td>
<td>Treats char as unsigned</td>
</tr>
</tbody>
</table>

\textit{Table 31: Compiler options summary}
### Summary of compiler options

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--clib</td>
<td>Uses the system include files for the CLIB library</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>--diagReminder</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</td>
<td>Uses the system include files for the DLIB library</td>
</tr>
<tr>
<td>--dlib_config</td>
<td>Uses the system include files for the DLIB library and determines which configuration of the library to use</td>
</tr>
<tr>
<td>--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>--eeec++</td>
<td>Enables Extended Embedded C++ syntax</td>
</tr>
<tr>
<td>--enable_multibyte</td>
<td>Enables support for multibyte characters in source files</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_r4</td>
<td>Excludes register R4 from use</td>
</tr>
<tr>
<td>--lock_r5</td>
<td>Excludes register R5 from use</td>
</tr>
<tr>
<td>--mfc</td>
<td>Enables multi-file compilation</td>
</tr>
</tbody>
</table>

Table 31: Compiler options summary (Continued)
<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--migration_preprocessor</td>
<td>Extends the preprocessor</td>
</tr>
<tr>
<td>--migration_preprocessor_extensions</td>
<td></td>
</tr>
<tr>
<td>--misrac</td>
<td>Enables error messages specific to MISRA-C:1998. This option is a synonym of</td>
</tr>
<tr>
<td></td>
<td>--misrac1998 and is only available for backwards compatibility.</td>
</tr>
<tr>
<td>--misrac1998</td>
<td>Enables error messages specific to MISRA-C:1998. See the IAR Embedded</td>
</tr>
<tr>
<td></td>
<td>Workbench® MISRA C Reference Guide.</td>
</tr>
<tr>
<td>--misrac2004</td>
<td>Enables error messages specific to MISRA-C:2004. See the IAR Embedded</td>
</tr>
<tr>
<td>--misrac_verbose</td>
<td>Enables verbose logging of MISRA C checking. See the IAR Embedded Workbench</td>
</tr>
<tr>
<td></td>
<td>® MISRA C Reference Guide or the IAR Embedded Workbench® MISRA C:2004 Reference Guide.</td>
</tr>
<tr>
<td>--module_name</td>
<td>Sets the object module name</td>
</tr>
<tr>
<td>--multiplier</td>
<td>Enables support for the hardware multiplier</td>
</tr>
<tr>
<td>--multiplier_location</td>
<td>Specifies the location of the hardware multiplier</td>
</tr>
<tr>
<td>--no_code_motion</td>
<td>Disables code motion optimization</td>
</tr>
<tr>
<td>--no_cse</td>
<td>Disables common subexpression elimination</td>
</tr>
<tr>
<td>--no_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_static_destruction</td>
<td>Disables destruction of C++ static variables at program exit</td>
</tr>
<tr>
<td>--no_system_include</td>
<td>Disables the automatic search for system include files</td>
</tr>
<tr>
<td>--no_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. Alias for --output.</td>
</tr>
<tr>
<td>--omit_types</td>
<td>Excludes type information</td>
</tr>
</tbody>
</table>

Table 31: Compiler options summary (Continued)
### Summary of Compiler Options

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>--pic</td>
<td>Produces position-independent code</td>
</tr>
<tr>
<td>--predef_macros</td>
<td>Lists the predefined symbols.</td>
</tr>
<tr>
<td>--preinclude</td>
<td>Includes an include file before reading the source file</td>
</tr>
<tr>
<td>--preprocess</td>
<td>Generates preprocessor output</td>
</tr>
<tr>
<td>--public_equ</td>
<td>Defines a global named assembler label</td>
</tr>
<tr>
<td>-r</td>
<td>Generates debug information. Alias for --debug.</td>
</tr>
<tr>
<td>--reduce_stack_usage</td>
<td>Reduces stack usage</td>
</tr>
<tr>
<td>--regvar_r4</td>
<td>Reserves register R4 for use by global register variables</td>
</tr>
<tr>
<td>--regvar_r5</td>
<td>Reserves register R5 for use by global register variables</td>
</tr>
<tr>
<td>--relaxed_fp</td>
<td>Relaxes the rules for optimizing floating-point expressions</td>
</tr>
<tr>
<td>--remarks</td>
<td>Enables remarks</td>
</tr>
<tr>
<td>--require_prototypes</td>
<td>Verifies that functions are declared before they are defined</td>
</tr>
<tr>
<td>--save_reg20</td>
<td>Declares all interrupt functions __save_reg20 by default</td>
</tr>
<tr>
<td>--segment</td>
<td>Changes a segment name</td>
</tr>
<tr>
<td>--silent</td>
<td>Sets silent operation</td>
</tr>
<tr>
<td>--strict</td>
<td>Checks for strict compliance with Standard C/C++</td>
</tr>
<tr>
<td>--system_include_dir</td>
<td>Specifies the path for system include files</td>
</tr>
<tr>
<td>--vla</td>
<td>Enables C99 VLA support</td>
</tr>
<tr>
<td>--warnings_affect_exit_code</td>
<td>Warnings affects exit code</td>
</tr>
<tr>
<td>--warnings_are_errors</td>
<td>Warnings are treated as errors</td>
</tr>
</tbody>
</table>

Table 31: Compiler options summary (Continued)
Descriptions of options

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

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

--c89

Syntax
--c89

Description
Use this option to enable the C89 standard instead of Standard C.

Note: This option is mandatory when the MISRA C checking is enabled.

See also
C language overview, page 105.

Project>Options>C/C++ Compiler>Language>C dialect>C89

--char_is_signed

Syntax
--char_is_signed

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

Note: The runtime library is compiled without the --char_is_signed option and cannot be used with code that is compiled with this option.

Project>Options>C/C++ Compiler>Language>Plain 'char' is

--char_is_unsigned

Syntax
--char_is_unsigned

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

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

Syntax

```
--clib
```

Description

Use this option to use the system header files for the CLIB library; the compiler will automatically locate the files and use them when compiling.

See also

```
--dlib, page 165 and --dlib_config, page 165.
```

To set related options, choose:

```
Project>Options>General Options>Library Configuration
```

--core

Syntax

```
--core={430|430X}
```

Parameters

```
430 (default) For devices based on the MSP430 architecture
430X For devices based on the MSP430X architecture
```

Description

Use this option to select the architecture for which the code is to be generated. If you do not use the option, the compiler generates code for the MSP430 architecture by default.

```
Project>Options>General Options>Target>Device
```

-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=

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

--data_model

Syntax

--data_model={small|medium|large}

Parameters

small (default) Sets data16 as the default memory type, both for data objects and pointers. This means that the first 64 Kbytes of memory can be accessed and that only 16-bit registers are used.

medium Sets data16 as the default memory type, both for data objects and pointers. This means that the first 64 Kbytes of memory can be accessed. But if required, also the entire memory can be used.

large Sets data20 as the default memory type, both for data objects and pointers. This means that the entire memory range can be accessed.

Description

For MSP430X devices, use this option to select the data model, which means a default placement of data objects. If you do not select a data model option, the compiler uses the default data model. Note that all modules of your application must use the same data model.

See also

Data models (MSP430X only), page 14.

Project>Options>General Options>Target>Data model
Descriptions of options

--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 154.

Description

Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension i.

Example

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

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

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

```
foo.r43: c:\iar\product\include\stdio.h
foo.r43: d:\myproject\include\foo.h
```

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

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

```
%.r43 : %.c

$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```
```
That is, in addition to producing an object file, the command also produces a dependency file in makefile style (in this example, using the extension `.d`).

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

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

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

   This option is not available in the IDE.

**--diag_error**

Syntax

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

Parameters

`tag` The number of a diagnostic message, for example the message number 177

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.

**Note:** By default, remarks are not displayed; use the `--diag_remark` option to display them.

**--diag_remark**

Syntax

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

Parameters

`tag` The number of a diagnostic message, for example the message number 177

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 `--diag_remark` option to display them.
Descriptions of options

--diag_suppress

Syntax

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

Parameters

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

Description

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

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

--diag_warning

Syntax

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

Parameters

tag     The number of a diagnostic message, for example the message number Pe826

Description

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

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

--diagnostics_tables

Syntax

--diagnostics_tables {filename|directory}

Parameters

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

Description

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

This option cannot be given together with other options.

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

Syntax
--discard_unused_publics

Description
Use this option to discard unused public functions and variables when compiling with the --mfc compiler option.

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

See also
--mfc, page 171 and Multi-file compilation units, page 135.

--dlib

Syntax
--dlib

Description
Use this option to use the system header files for the DLIB library; the compiler will automatically locate the files and use them when compiling.

Note: The DLIB library is used by default: To use the CLIB library, use the --clib option instead.

See also
--dlib_config, page 165, --no_system_include, page 175, --system_include_dir, page 186, and --clib, page 160.

To set related options, choose:
Project>Options>General Options>Library Configuration

--dlib_config

Syntax
--dlib_config filename.h|none

Parameters

filename
A DLIB configuration header file. For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 154.

none
No configuration file will be used.
Each runtime library has a corresponding library configuration file. Use this option to explicitly specify which library configuration file to use, either by specifying an explicit file or by specifying no library configuration. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `430\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 47.

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 56.

**Note:** This option only applies to the IAR DLIB runtime environment.

To set related options, choose:

*Project*-*Options*-*General Options*-*Library Configuration*

---

### --double

**Syntax**

```
--double={32|64}
```

**Parameters**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</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 193.

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

---
**-e**

**Syntax**

-e

**Description**

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

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

**See also**

Enabling language extensions, page 107.

**--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++.

**Note:** By default, this option is selected in the IDE.

**--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 114.
--enable_multibytes

Syntax
--enable_multibytes

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

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

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

--error_limit

Syntax
--error_limit=n

Parameters

\begin{tabular}{l}
\textit{n} & The number of errors before the compiler stops the compilation. \\
& \textit{n} must be a positive integer; 0 indicates no limit.
\end{tabular}

Description
Use the \texttt{--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 \textit{filename}

Parameters

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

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

---

### --header_context

**Syntax**

```bash
--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**

```bash
-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 148.

---

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

---

### -l

**Syntax**

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

**Parameters**

- **a** (default)
  
  Assembler list file

- **A**
  
  Assembler list file with C or C++ source as comments
Descriptions of options

---

**b** Basic assembler list file. This file has the same contents as a list file produced with `--la`, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included.

**B** Basic assembler list file. This file has the same contents as a list file produced with `--la`, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included.

**c** C or C++ list file

**C** (default) C or C++ list file with assembler source as comments

**D** C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values

**N** No diagnostics in file

**H** Include source lines from header files in output. Without this option, only source lines from the primary source file are included.

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

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

**Description**

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

**To set related options, choose:**

`Project>Options>C/C++ Compiler>List`

**--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.

**To set related options, choose:**

`Project>Options>C/C++ Compiler>Output>Module type>Library Module`
--lock_r4
Syntax --lock_R4
Description Use this option to exclude the register R4 from use by the compiler. This makes the module linkable with both modules that use R4 as __regvar and modules that does not define its R4 usage. Use this option if the R4 registers is used by another tool, for example a ROM-monitor debugger.

Project>Options>C/C++ Compiler>Code>R4 utilization>Not used

--lock_r5
Syntax --lock_R5
Description Use this option to exclude the register R5 from use by the compiler. This makes the module linkable with both modules that use R5 as __regvar and modules that does not define its R5 usage. Use this option if the R5 registers is used by another tool, for example a ROM-monitor debugger.

Project>Options>C/C++ Compiler>Code>R5 utilization>Not used

--mfc
Syntax --mfc
Description Use this option to enable multi-file compilation. This means that the compiler compiles one or several source files specified on the command line as one unit, which enhances interprocedural optimizations.

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

Example icc430 myfile1.c myfile2.c myfile3.c --mfc
See also --discard_unused_publics, page 165, --output, -o, page 179, and Multi-file compilation units, page 135.

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

Syntax

--migration_preprocessor_extensions

Description

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

● Floating-point expressions
● Basic type names and sizeof
● All symbol names (including typedefs and variables).

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

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

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

--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
--multiplier

Syntax

--multiplier\=[16|16s|32]\]

Parameters

- No parameter: The traditional hardware multiplier
- 16: The traditional hardware multiplier
- 16s: The extended hardware multiplier used by some 2xx devices
- 32: The 32-bit hardware multiplier

Description

Use this option to generate code that accesses the hardware multiplier. This will also disable interrupts.

Note: You should also redirect library function calls to variants that use the hardware multiplier, see Hardware multiplier support, page 72.

To set related options, choose:

Project>Options>General Options>Target>Device

and

Project>Options>General Options>Target>Hardware multiplier

--multiplier_location

Syntax

--multiplier_location=address

Parameters

- address: The address of where the multiplier is located.

Description

For some MSP430 devices, the compiler must know the location of the hardware multiplier to generate code that makes use of it. Use this option to inform the compiler of where the hardware multiplier is located.

You must also specify the --multiplier option to use this option.

Example

Some 5xx devices need the following option:

--multiplier_location=40

To set related options, choose:

Project>Options>General Options>Target>Device
**--no_code_motion**

**Syntax**

```
--no_code_motion
```

**Description**

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

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

![Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion](image)

**--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.

![Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination](image)

**--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, but the resulting code might be difficult to debug.

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

If you do not want to disable inlining for a whole module, use `#pragma inline=never` on an individual function instead.
Note: This option has no effect at optimization levels below High.

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

**--no_path_in_file_macros**

Syntax  
```
--no_path_in_file_macros
```

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

See also  
*Descriptions of predefined preprocessor symbols*, page 242.

This option is not available in the IDE.

**--no_static_destruction**

Syntax  
```
--no_static_destruction
```

Description  
Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed. Use this option to suppress the emission of such code.

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

**--no_system_include**

Syntax  
```
--no_system_include
```

Description  
By default, the compiler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the `-I` compiler option.

See also  
`--dlib`, page 165, `--dlib_config`, page 165, and `--system_include_dir`, page 186.

To set this option, use Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories.
--no_tbaa

Syntax  
   --no_tbaa

Description  
Use this option to disable type-based alias analysis. 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 138.

--no_typedefs_in_diagnostics

Syntax  
   --no_typdefs_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
d               debug.
               The compiler heuristically decides which loops to unroll. Different heuristics are used
               when optimizing for speed and size.
               Note: This option has no effect at optimization levels below High.
               Project>Options>C/C++ Compiler>Optimizations>Enable
               transformations>Loop unrolling

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

--no_wrap_diagnostics
Syntax        --no_wrap_diagnostics
Description   By default, long lines in diagnostic messages are broken into several lines to make the
               message easier to read. Use this option to disable line wrapping of diagnostic messages.
               This option is not available in the IDE.
Descriptions of options

---

-O

Syntax

-0[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 135.

--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.

IAR C/C++ Compiler

Reference Guide for MSP430

178
Compiler options

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

This option is not available in the IDE.

--output, -o
Syntax: --output {filename|directory} -o {filename|directory}
Parameters: For information about specifying a filename or a directory, see Rules for specifying a filename or directory as parameters, page 154.
Description: By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension .r43. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--pic
Syntax: --pic
Description: Use this option to make the compiler produce position-independent code, which means that functions can be relocated at runtime.

This option is not available when compiling for the MSP430X architecture.

Project>Options>General Options>Target>Position-independent code

--predef_macros
Syntax: --predef_macros {filename|directory}
Parameters: For information about specifying a filename, see Rules for specifying a filename or directory as parameters, page 154.
### Description

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

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

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

This option is not available in the IDE.

#### --preinclude

**Syntax**

```bash
--preinclude includefile
```

**Parameters**

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

**Description**

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

#### --preprocess

**Syntax**

```bash
--preprocess [=c][n][l] (filename|directory)
```

**Parameters**

- `c`  
  Preserve comments
- `n`  
  Preprocess only
- `l`  
  Generate `#line` directives

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

**Description**

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

---

### --public_equ

**Syntax**

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

**Parameters**

- `symbol`  
  The name of the assembler symbol to be defined

- `value`  
  An optional value of the defined assembler symbol

**Description**

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

This option is not available in the IDE.

---

### --reduce_stack_usage

**Syntax**

```
--reduce_stack_usage
```

**Description**

Use this option to make the compiler minimize the use of stack space at the cost of somewhat larger and slower code.

**See also**  
Project>Options>C/C++ Compiler>Code>Reduce stack usage

---

### --regvar_r4

**Syntax**

```
--regvar_r4
```

**Description**

Use this option to reserve the register R4 for use by global register variables, declared with the `__regvar` attribute. This can give more efficient code if used on frequently used variables.

**See also**  
--lock_r4, page 171 and __regvar, page 209.

**See also**  
Project>Options>C/C++ Compiler>Code>R4 utilization>__regvar variables
--regvar_r5

Syntax
--regvar_R5

Description
Use this option to reserve the register R5 for use by global register variables, declared with the __regvar attribute. This can give more efficient code if used on frequently used variables.

See also
--lock_r5, page 171 and __regvar, page 209.

Project>Options>C/C++ Compiler>Code>R5 utilization>__regvar variables

--relaxed_fp

Syntax
--relaxed_fp

Description
Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:
- The expression consists of both single- and double-precision values
- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

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

Example
float f(float a, float b)
{"return a + b * 3.0;"}

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

Project>Options>General Options>Language>Relaxed floating-point precision
--- remarks

Syntax: `--remarks`

Description: The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also: "Severity levels", page 151.

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

--- require_prototypes

Syntax: `--require_prototypes`

Description: Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

Project>Options>C/C++ Compiler>Language>Require prototypes

--- save_reg20

Syntax: `--save_reg20`

Description: Use this option to make all interrupt functions be treated as a __save_reg20 declared function. This means that you do not have to explicitly use the __save_reg20 keyword on any interrupt functions.

This is necessary if your application requires that all 20 bits of registers are preserved. The drawback is that the code will be somewhat slower.

Note: This option is only available when compiling for the MSP430X architecture.
See also

__save_reg20, page 211

Project>Options>C/C++ Compiler>Code>20-bit context save on interrupt

--segment

Syntax

--segment {memory_attribute|segment_type}=new_segment_name

Parameters

 memory_attribute: One of the following memory attributes:
   __data16
   __data20
   __regvar

 segment_type: One of the following segment types (in lower-case letters):
   cstack
   code
   difunct
   intvec
   isr_code
   ramfunc_code

new_segment_name: If a memory attribute was specified to the left of the equal sign, this is
the new name for the segment group. If a segment type was specified to the left of the equal
sign, this is the new name of the segment used for the type.

Description

The compiler places functions and data objects into named segments which are referred to by the IAR XLINK Linker. Use the --segment option to perform one of these operations:

- Place all functions or data objects declared with the __memory_attribute in segments with names that begin with new_segment_name
- Rename a segment.

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

Example 1

This command places the __data16 int a; declared variable in the MYDATA_Z segment:

--segment __data16=MYDATA
Example 2

This command changes the name of the INTVEC segment to MYINTS:

```
--segment intvec=MYINTS
```

See also

*Controlling data and function placement in memory, page 131* and *Summary of extended keywords, page 204.*

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

---

**--silent**

**Syntax**

```
--silent
```

**Description**

By default, the compiler issues introductory messages and a final statistics report. Use this option to make the compiler operate without sending these messages to the standard output stream (normally the screen).

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

This option is not available in the IDE.

---

**--strict**

**Syntax**

```
--strict
```

**Description**

By default, the compiler accepts a relaxed superset of Standard C and C++. Use this option to ensure that the source code of your application instead conforms to strict Standard C and C++.

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

See also

*Enabling language extensions, page 107.*

Project>Options>C/C++ Compiler>Language>Language conformance>Strict
Descriptions of options

--system_include_dir
Syntax
--system_include_dir path
Parameters
path

Description
By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

See also
--dlib, page 165, --dlib_config, page 165, and --no_system_include, page 175.

This option is not available in the IDE.

--vla
Syntax
--vla
Description
Use this option to enable support for C99 variable length arrays. Note that this option requires Standard C and cannot be used together with the --c89 compiler option.

See also
C language overview, page 105.

Project>Options>C/C++ Compiler>Language>C dialect>Allow VLA

--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

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 164.

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

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

See the chapter Efficient coding for embedded applications for information about which data types 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.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure. For more information about pad bytes, see Packed structure types, page 197.

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

ALIGNMENT ON THE MSP430 MICROCONTROLLER

The MSP430 microcontroller can access memory using 8- or 16-bit accesses. However, when a 16-bit access is performed, the data must be located at an even address. The
The compiler ensures this by assigning an alignment to every data type, which means that the MSP430 microcontroller can read the data.

## Basic data types

The compiler supports both all Standard C basic data types and some additional types.

### INTEGER TYPES

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

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

Table 32: Integer types

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

### Bool

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

### The long long type

The `long long` data type is supported with these restrictions:

- The CLIB runtime library does not support the `long long` type
- 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.

Note: The IAR CLIB Library has only rudimentary support for wchar_t.

Bitfields

In Standard C, int, signed int, and unsigned int can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (char, short, int, etc) results in a signed or unsigned bitfield.

Bitfields in expressions will have the same data type as the integer base type.

In the IAR C/C++ Compiler for MSP430, plain integer types are treated as unsigned.

Bitfields in expressions are treated as int if int can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.
Each bitfield is placed into a container of its base type from the least significant bit to
the most significant bit. If the last container is of the same type and has enough bits
available, the bitfield is placed into this container, otherwise a new container is
allocated.

If you use the directive `#pragma bitfield=reversed`, bitfields are placed from the
most significant bit to the least significant bit in each container. See `bitfields`, page 216.

**Example**

Assume this example:

```c
struct bitfield_example
{   uint32_t a : 12;
    uint16_t b : 3;
    uint16_t c : 7;
    uint8_t  d;
};
```

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset 0 and puts
`a` into the least significant 12 bits of the container.

To place the second bitfield, `b`, a new container is allocated at offset 4, because the type
of the bitfield is not the same as that of the previous one. `b` is placed into the least
significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

The fourth member, `d`, is allocated into the byte at offset 6. `d` cannot be placed into the
same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it
would not fit.

When using reverse order, each bitfield is instead placed starting from the most
significant bit of its container.
This is the layout of `bitfield_example`:

![Diagram of `bitfield_example` layout](image)

**FLOATING-POINT TYPES**

In the IAR C/C++ Compiler for MSP430, 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 <code>--double=32</code></th>
<th>Size if <code>--double=64</code></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 33: Floating-point types*

**Note:** The size of `double` and `long double` depends on the `--double=({32|64})` option, see `--double`, page 166. The type `long double` uses the same precision as `double`.

The compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero.

Exception flags according to the IEEE 754 standard are not supported.
**32-bit floating-point format**

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

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

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 at least:
\[\pm 1.18\times 10^{-38} \text{ to } \pm 3.39 \times 10^{38}\]

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:

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

The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:
\[(-1)^S \times 2^{(\text{Exponent}-1023)} \times 1.\text{Mantissa}\]

The range of the number is at least:
\[\pm 2.23 \times 10^{-308} \text{ to } \pm 1.79 \times 10^{308}\]

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

**Representation of special floating-point numbers**

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

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.

Note: The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, NaN. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

### Pointer types

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

#### FUNCTION POINTERS

For the MSP430 architecture, function pointers are always 16 bits. For the MSP430X architecture, function pointers are 20 bits and they can address the entire 1 Mbyte of memory. See this table:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Data model</th>
<th>In register</th>
<th>In memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP430</td>
<td>—</td>
<td>One 16-bit register</td>
<td>2 bytes</td>
</tr>
<tr>
<td>MSP430X</td>
<td>Small</td>
<td>Two 16-bit registers</td>
<td>4 bytes</td>
</tr>
<tr>
<td>MSP430X</td>
<td>Medium</td>
<td>One 20-bit register</td>
<td>4 bytes</td>
</tr>
<tr>
<td>MSP430X</td>
<td>Large</td>
<td>One 20-bit register</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

*Table 34: Function pointers*

#### DATA POINTERS

These data pointers are available:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Pointer size</th>
<th>Index type</th>
<th>Address range</th>
</tr>
</thead>
<tbody>
<tr>
<td>__data16</td>
<td>16 bits</td>
<td>signed int</td>
<td>0x0–0xFFFF</td>
</tr>
<tr>
<td>_data20*</td>
<td>20 bits</td>
<td>signed int</td>
<td>0x0–0xFFFFF</td>
</tr>
</tbody>
</table>

*Table 35: Data pointers*

*The _data20* pointer type is not available in the Small data model.

#### CASTING

Casts between pointers have these characteristics:

- Casting a value of an integer type to a pointer of a smaller type is performed by truncation
- Casting a value of an integer type to a pointer of a larger type is performed by zero extension
Structure types

- Casting a pointer type to a smaller integer type is performed by truncation
- Casting a pointer type to a larger integer type is performed by zero extension
- Casting a data pointer to a function pointer and vice versa is illegal
- Casting a function pointer to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension
- Casting from a larger pointer to a smaller pointer is performed by truncation.

size_t

size_t is the unsigned integer type required to hold the maximum size of an object. For the MSP430 architecture, and for the MSP430X architecture in the Small and Medium data models, the size of size_t is 16 bits. In the Large data model, the size of size_t is 32 bits.

ptrdiff_t

ptrdiff_t is the type of the signed integer required to hold the difference between two pointers to elements of the same array. For the MSP430 architecture, and for the MSP430X architecture in the Small and Medium data models, the size of ptrdiff_t is 16 bits. In the Large data model, the size of ptrdiff_t is 32 bits.

Note: Subtracting the start address of an object from the end address can yield a negative value, because the object can be larger than what the ptrdiff_t can represent. See this example:

```c
char buff[60000];  /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff;   /* signed integer type. */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;
```

intptr_t

intptr_t is a signed integer type large enough to contain a void *. For the MSP430 architecture, and for the MSP430X architecture in the Small and Medium data models, the size of intptr_t is 16 bits. In the Large data model, the size of intptr_t is 32 bits.

uintptr_t

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

Structure types

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

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

GENERAL LAYOUT

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

Example

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

The alignment of the structure is 1 byte, and the size is 3 bytes.

![Figure 6: 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 #pragma pack directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

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

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

This example declares a packed structure:

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

In this example, the structure `S` has this memory layout:

![Figure 7: Packed structure layout](image)

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

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

`S2` has this memory layout:

![Figure 8: Packed structure layout](image)

The structure `S` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 2, which means that alignment of the structure `S2` will become 2.

For more information, see *Alignment of elements in a structure*, page 129.
Type qualifiers

According to the C standard, volatile and const are type qualifiers.

DECLARING OBJECTS VOLATILE

By declaring an object volatile, the compiler is informed that the value of the object can change beyond the compiler’s control. The compiler must also assume that any accesses can have side effects—thus all accesses to the volatile object must be preserved.

There are three main reasons for declaring an object volatile:

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

Definition of access to volatile objects

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

● The compiler considers each read and write access to an object declared volatile as an access
● The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:
  char volatile a;
  a = 5;  /* A write access */
  a += 6; /* First a read then a write access */
● An access to a bitfield is treated as an access to the underlying type
● Adding a const qualifier to a volatile object will make write accesses to the object impossible. However, the object will be placed in RAM as specified by the C standard.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for MSP430 are described below.
**Rules for accesses**

In the IAR C/C++ Compiler for MSP430, accesses to `volatile` declared objects are subject to these rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The compiler adheres to these rules for all 8-bit and 16-bit memory accesses. For MSP430X also for all 20-bit memory accesses.

For larger types, all accesses are preserved but it is not guaranteed that all parts of the object is accessed.

**DECLARING OBJECTS VOLATILE AND CONST**

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, declare it with the `_ro_placement` attribute. See `_ro_placement`, page 210.

**DECLARING OBJECTS CONST**

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

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

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

---

**Data types in C++**

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

This chapter describes the extended keywords that support specific features of the MSP430 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 MSP430 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 detailed information about each attribute, see Descriptions of extended keywords, page 205.

Note: The extended keywords are only available when language extensions are enabled in the compiler.

In the IDE, language extensions are enabled by default.

Use the -e compiler option to enable language extensions. See -e, page 167 for additional information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive #pragma type_attribute.
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 data memory attributes: __data16, __data20, and __regvar.

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive #pragma type_attribute, an appropriate default attribute is used. You can specify one memory attribute for each level of pointer indirection.

**General type attributes**

These general type attributes are available:

- **Function type attributes** affect how the function should be called: __interrupt, __monitor, __task, __cc_version1, and __cc_version2
- **Data type attributes**: const 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 199.

**Syntax for type attributes used on data objects**

In general, type attributes for data objects follow the same syntax as the type qualifiers const and volatile.

The following declaration assigns the __data20 type attribute to the variables i and j; in other words, the variable i and j is placed in data20 memory. The variables k and l behave in the same way:

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

Note that the attribute affects both identifiers.

This declaration of i and j is equivalent with the previous one:

```c
#pragma type_attribute=__data20
int i, j;
```
The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Note that the pragma directive has no effect if a memory attribute is already explicitly declared.

For more examples of using memory attributes, see More examples, page 19.

An easier way of specifying storage is to use type definitions. These two declarations are equivalent:

```c
typedef char __data20 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
and
__data20 char b;
char __data20 *bp;
```

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

**Syntax for type attributes on data pointers**

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

```c
int __data20 * p;  // The int object is located in __data20 memory.
int * __data20 p;  // The pointer is located in __data20 memory.
__data20 int * p;  // The pointer is located in __data20 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);
```
OBJECT ATTRIBUTES

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object. Any exceptions to this rule are noted in the description of the attribute.

These object attributes are available:

- Object attributes that can be used for variables: `__no_init`, `__persistent`, `__ro_placement`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__noreturn`, `__ramfunc`, `__raw`, `__save_reg20`, 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 131. For more information about `vector`, see `vector`, page 230.

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><code>__cc_version1</code></td>
<td>Specifies the Version1 calling convention</td>
</tr>
<tr>
<td><code>__cc_version2</code></td>
<td>Specifies the Version2 calling convention</td>
</tr>
</tbody>
</table>

Table 36: Extended keywords summary
## Descrpitions of extended keywords

These sections give detailed information about each extended keyword.

### __cc_version1

<table>
<thead>
<tr>
<th>Extended keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__data16</td>
<td>Controls the storage of data objects</td>
</tr>
<tr>
<td>__data20</td>
<td>Controls the storage of data objects</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 function will not return</td>
</tr>
<tr>
<td>__persistent</td>
<td>Ensures that variables are only initialized, for example, by a code downloader, and not by cstartup.</td>
</tr>
<tr>
<td>__ramfunc</td>
<td>Makes a function execute in RAM</td>
</tr>
<tr>
<td>__raw</td>
<td>Prevents saving used registers in interrupt functions</td>
</tr>
<tr>
<td>__regvar</td>
<td>Permanently places a variable in a specified register</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>__ro_placement</td>
<td>Places const volatile data in read-only memory</td>
</tr>
<tr>
<td>__save_reg20</td>
<td>Saves and restores all 20 bits in 20-bit registers</td>
</tr>
<tr>
<td>__task</td>
<td>Relaxes the rules for preserving registers</td>
</tr>
</tbody>
</table>

*Table 36: Extended keywords summary (Continued)*

### Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 201.

### Description

The __cc_version1 keyword is available for backward compatibility of the interface for calling assembler routines from C. It makes a function use the calling convention of the IAR C/C++ Compiler for MSP430 version 1.x–3.x instead of the default calling convention.

### Example

```
__cc_version1 int func(int arg1, double arg2)
```

### See also

*Calling convention*, page 91.
Descriptions of extended keywords

__cc_version2

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 201.

Description

The __cc_version2 keyword sets the default calling convention of the interface for calling assembler routines from C. It makes a function use the calling convention of the IAR C/C++ Compiler for MSP430 version 4.x and later.

Example

__cc_version2 int func(int arg1, double arg2)

See also

Calling convention, page 91.

__data16

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see Type attributes, page 201.

Description

The __data16 memory attribute overrides the default storage of variables and places individual variables and constants in data16 memory, which is the entire 64 Kbytes of memory in the MSP430 architecture and the lower 64 Kbytes in the MSP430X architecture. You can also use the __data16 attribute to create a pointer explicitly pointing to an object located in the data16 memory.

Storage information

- Address range: 0–0xFFFF (64 Kbytes)
- Maximum object size: 65535 bytes
- Pointer size: 2 bytes

Example

__data16 int x;

See also

Memory types (MSP430X only), page 15.

__data20

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see Type attributes, page 201.

Description

The __data20 memory attribute overrides the default storage of variables and places individual variables and constants in data20 memory, which is the entire 1 Mbyte of memory in the MSP430X architecture. You can also use the __data20 attribute to create a pointer explicitly pointing to an object located in the data20 memory.
The __data20 attribute cannot be used in the Small data model.

Storage information
- Address range: 0–0xFFFFF (1 Mbyte)
- Maximum object size: 1 Mbyte
- Pointer size: 20 bits in register, 4 bytes in memory

Example
__data20 int x;

See also Memory types (MSP430X only), page 15.

__interrupt

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 201.

Description
The __interrupt keyword specifies interrupt functions. To specify one or several interrupt vectors, use the #pragma vector directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a void return type and cannot have any parameters.

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

Example
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);

See also Interrupt functions, page 23, vector, page 230, INTVEC, page 268.

__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 201.

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**

```c
__monitor int get_lock(void);
```

**See also**


**__no_init**

**Syntax**

Follows the generic syntax rules for object attributes, see Object attributes, page 204.

**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**

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

**__noreturn**

**Syntax**

Follows the generic syntax rules for object attributes, see Object attributes, page 204.

**Description**

The __noreturn keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are abort and exit.

**Example**

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

**__persistent**

**Syntax**

Follows the generic syntax rules for object attributes, see Object attributes, page 204.

**Description**

Global or static variables defined with the __persistent attribute will not be initialized by cstartup, but only by a code downloader or similar add-on functionality. This attribute cannot be used together with any of the keywords const or __no_init, and __persistent data cannot be located to an absolute address (using the @ operator).

**Example**

```c
__persistent int x = 0;
```

**See also**

__ramfunc
Syntax
Follows the generic syntax rules for object attributes, see Object attributes, page 204.

Description
The __ramfunc keyword makes a function execute in RAM. Two code segments will be created: one for the RAM execution, and one for the ROM initialization.
Functions declared __ramfunc are by default stored in the CODE_I segment.

Example
__ramfunc int FlashPage(char * data, char * page);

See also
Execution in RAM, page 30.

__raw
Syntax
Follows the generic syntax rules for object attributes, see Object attributes, page 204.

Description
This keyword prevents saving used registers in interrupt functions.
Interrupt functions preserve the contents of all used processor registers at function entrance and restore them at exit. However, for some very special applications, it can be desirable to prevent the registers from being saved at function entrance. This can be accomplished by the use of the keyword __raw.

Example
__raw __interrupt void my_interrupt_function()

__regvar
Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see Type attributes, page 201.

Description
This keyword is used for declaring that a global or static variable should be placed permanently in the specified register. The registers R4–R5 can be used for this purpose, provided that they have been reserved with one of the --regvar_r4 or --regvar_r5 compiler options.

The __regvar attribute can be used on integer types, pointers, 32-bit floating-point numbers, structures with one element and unions of all these. However, it is not possible to point to an object that has been declared __regvar. An object declared __regvar cannot have an initial value.

Note: If a module in your application has been compiled using --regvar_r4, it can only be linked with modules that have been compiled with either --regvar_r4 or --lock_r4. The same is true for --regvar_r5/--lock_r5.
Example

To declare a global register variable, use the following syntax:

```
__regvar __no_init type variable_name @ location
```

where `location` is either `__R4` or `__R5`, declared in `intrinsics.h`.

This will create a variable called `variable_name` of type `type`, located in register `R4` or `R5`, for example:

```
__regvar __no_init int counter @ __R4;
```

See also

`--regvar_r4`, page 181 and `--regvar_r5`, page 182.

__(root)

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 204.

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*.

__(ro_placement)

Syntax

Although this is an object attribute, it follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 201. Just like a type attribute, it must be specified both when a data object is defined and when it is declared.

Description

Use the `__ro_placement` attribute in combination with the type qualifiers `const` and `volatile` to inform the compiler that a variable should be placed in read-only (code) memory.

This is useful, for example, for placing a `const volatile` variable in flash memory residing in ROM.

Example

```
__ro_placement const volatile int x = 10;
```
**__save_reg20**

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 204.

Description

When compiling for the MSP430X architecture in the Small data model, use this keyword to save and restore all 20 bits of the registers that are used, instead of only 16 bits, which are saved and restored by normal functions. This keyword will make the function save all registers and not only the ones used by the function to guarantee that 20-bit registers are not destroyed by subsequent calls.

This may be necessary if the function is called from assembler routines that use the upper 4 bits of the 20-bit registers.

**Note:** The `__save_reg20` keyword has only effect when compiling for the MSP430X architecture.

Example

```c
__save_reg20 void myFunction(void);
```

See also


**__task**

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 201.

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);
```
Descriptions of extended keywords
Pragma directives

This chapter describes the pragma directives of the compiler.

The #pragma directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the #pragma preprocessor directive or the _Pragma() preprocessor operator:

<table>
<thead>
<tr>
<th>Pragma directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_template_matching</td>
<td>Makes a template function fully memory-attribute aware</td>
</tr>
<tr>
<td>bis_nmi_iel</td>
<td>Generates a BIS instruction just before the RETI instruction</td>
</tr>
<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>dataseg</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>error</td>
<td>Signals an error while parsing</td>
</tr>
<tr>
<td>include_alias</td>
<td>Specifies an alias for an include file</td>
</tr>
<tr>
<td>inline</td>
<td>Controls inlining of a function</td>
</tr>
</tbody>
</table>

Table 37: Pragma directives summary
## Summary of pragma directives

<table>
<thead>
<tr>
<th>Pragma directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>no_epilogue</td>
<td>Performs a local return sequence</td>
</tr>
<tr>
<td>object_attribute</td>
<td>Changes the definition of a variable or a function</td>
</tr>
<tr>
<td>optimize</td>
<td>Specifies the type and level of an optimization</td>
</tr>
<tr>
<td>pack</td>
<td>Specifies the alignment of structures and union members</td>
</tr>
<tr>
<td>__printf_args</td>
<td>Verifies that a function with a printf-style format string is called with the correct arguments</td>
</tr>
<tr>
<td>required</td>
<td>Ensures that a symbol that is needed by another symbol is included in the linked output</td>
</tr>
<tr>
<td>rtmodel</td>
<td>Adds a runtime model attribute to the module</td>
</tr>
<tr>
<td>__scanf_args</td>
<td>Verifies that a function with a scanf-style format string is called with the correct arguments</td>
</tr>
<tr>
<td>section</td>
<td>This directive is an alias for #pragma segment</td>
</tr>
<tr>
<td>segment</td>
<td>Declares a segment name to be used by intrinsic functions</td>
</tr>
<tr>
<td>STDC CX_LIMITED_RANGE</td>
<td>Specifies whether the compiler can use normal complex mathematical formulas or not</td>
</tr>
<tr>
<td>STDC FENV_ACCESS</td>
<td>Specifies whether your source code accesses the floating-point environment or not.</td>
</tr>
<tr>
<td>STDC FP_CONTRACT</td>
<td>Specifies whether the compiler is allowed to contract floating-point expressions or not.</td>
</tr>
<tr>
<td>type_attribute</td>
<td>Changes the declaration and definitions of a variable or function</td>
</tr>
<tr>
<td>vector</td>
<td>Specifies the vector of an interrupt or trap function</td>
</tr>
</tbody>
</table>

Table 37: Pragma directives summary (Continued)

Note: For portability reasons, see also Recognized pragma directives (6.10.6), page 276.
Descriptions of pragma directives

This section gives detailed information about each pragma directive.

basic_template_matching

Syntax

```c
#pragma basic_template_matching
```

Description

Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications described in Templates and data memory attributes, page 120.

Example

```c
#pragma basic_template_matching
template<typename T> void fun(T *);
fun((int __data16 *) 0); /* Template parameter T becomes int __data16 */
```

bis_nmi_ie1

Syntax

```c
#pragma bis_nmi_ie1=mask
```

Parameters

```c
mask A constant expression
```

Description

Use this pragma directive for changing the interrupt control bits in the register IE, within an NMI service routine. A `BIS.W #mask, IE1` instruction is generated immediately before the `RETI` instruction at the end of the function, after any POP instructions.

The effect is that NMI interrupts cannot occur until after the BIS instruction. The advantage of placing it at the end of the POP instructions is that less stack will be used in the case of nested interrupts.

Example

In the following example, the OFIE bit will be set as the last instruction before the RETI instruction:

```c
#pragma bis_nmi_ie1=OFIE
#pragma vector=NMI_VECTOR
__interrupt void myInterruptFunction(void)
{
...
}
```
Descriptions of pragma directives

### bitfields

**Syntax**

```
#pragma bitfields={reversed|default}
```

**Parameters**

- **reversed**
  - Bitfield members are placed from the most significant bit to the least significant bit.
- **default**
  - Bitfield members are placed from the least significant bit to the most significant bit.

**Description**

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

**Example**

```
/* Structure that uses reversed bitfields. */
{
  unsigned char error :1;
  unsigned char size  :4;
  unsigned short code :10;
}
#pragma bitfields=default
/* Restores to default setting. */
```

**See also**

*Bitfields*, page 191.

### constseg

**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**

This legacy pragma directive is supported for backward compatibility reasons. It can be used 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=__data20 MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```
Pragma directives

**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**

This legacy pragma directive is supported for backward compatibility reasons. It can be used 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=\_\_data20 MY_SEGMENT
\_\_no_init char myBuffer[1000];
#pragma dataseg=default
```
**diag_default**

Syntax

```c
#pragma diag_default=tag[,tag,...]
```

Parameters

- **tag**
  - The number of a diagnostic message, for example the message number Pe117.

Description

Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also

*Diagnostics*, page 151.

**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 151.

**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 151.
Pragma directives

**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 151.

**diag_warning**

**Syntax**

```c
#pragma diag_warning=tag[,tag,...]
```

**Parameters**

- `tag`
  
  The number of a diagnostic message, for example the message number Pe826.

**Description**

Use this pragma directive to change the severity level to warning for the specified diagnostic messages.

**See also**

*Diagnostics*, page 151.

**error**

**Syntax**

```c
#pragma error message
```

**Parameters**

- `message`
  
  A string that represents the error message.

**Description**

Use this pragma directive to cause an error message when it is parsed. This mechanism is different from the preprocessor directive `#error`, because the `#pragma error` directive can be included in a preprocessor macro using the `_Pragma` form of the directive and only causes an error if the macro is used.

**Example**

```c
#if FOO_AVAILABLE
#define FOO ...
#else
#define FOO _Pragma("error\"Foo is not available\"")
#endif
```
If `Foo_AVAILABLE` is zero, an error will be signaled if the `Foo` macro is used in actual source code.

### include_alias

**Syntax**

```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>)
#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 148.

### inline

**Syntax**

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

**Parameters**

- **No parameter**: Has the same effect as the `inline` keyword.
- **forced**: Disables the compiler’s heuristics and forces inlining.
- **never**: Disables the compiler’s heuristics and makes sure that the function will not be inlined.

**Description**

Use `#pragma inline` to advise the compiler that the function 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.
Pragma directives

#pragma inline is similar to the C++ keyword inline. The difference is that the compiler uses C++ inline semantics for the #pragma inline directive, but uses the Standard C semantics for the inline keyword.

Specifying #pragma inline=never disables the compiler’s heuristics and makes sure that the function will not be inlined.

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.

See also

Function inlining, page 137.

language

Syntax

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

Parameters

extended Enables the IAR Systems language extensions from the first use of the pragma directive and onward.

default From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.

save|restore Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code. Each use of save must be followed by a matching restore in the same file without any intervening #include directive.

Description

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

Example 1

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

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

Example 2

Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled:

```c
#pragma language=extended
/* Part of source code. */
#pragma language=default
```

Example 3

Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled, but where the state before the sequence cannot be assumed to be the same as that specified by the compiler options in use:

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

See also

- `-c`, page 167 and `--strict`, page 185.

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

```c
#pragma location=0x22E
__no_init volatile char PORT1; /* PORT1 is located at address 0x22E */

#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 131.

message

Syntax

#pragma message(message)

Parameters

message The message that you want to direct to the standard output stream.

Description

Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example:

#ifdef TESTING
#pragma message("Testing")
#endif

no_epilogue

Syntax

#pragma no_epilogue

Description

Use this pragma directive to use a local return sequence instead of a call to the library routine EpilogueN. This pragma directive can be used when a function needs to exist on its own as in for example a bootloader that needs to be independent of the libraries it is replacing.

Example

#pragma no_epilogue
void bootloader(void) @$BOOTSECTOR$
{
...

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 204.
Descriptions of pragma directives

**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 201.

---

**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.
Pragma directives

Example

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

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

pack

Syntax

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

Parameters

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

Description

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

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

Note that accessing an object that is not aligned at its correct alignment requires code that is both larger and slower than the code needed to access the same kind of object when aligned correctly. If there are many accesses to such fields in the program, it is usually better to construct the correct values in a struct that is not packed, and access this instead.

Also, special care is needed when creating and using pointers to misaligned fields. For direct access to misaligned fields in a packed struct, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned field is accessed...
through a pointer to the field, the normal (smaller and faster) code for accessing the type of the field is used. In the general case, this will not work.

See also

*Structure types*, page 196.

---

**__printf_args**

**Syntax**

`#pragma __printf_args`

**Description**

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

**Example**

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

/* Function call */
printf("%d",x); /* Compiler checks that x is an integer */
```

---

**required**

**Syntax**

`#pragma required=symbol`

**Parameters**

```
symbol
```

Any statically linked function or variable.

**Description**

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

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()
{
   /* Do something here. */
}
```
Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

### rtmodel

**Syntax**

```
#pragma rtmodel="key","value"
```

**Parameters**

- **"key"**
  A text string that specifies the runtime model attribute.

- **"value"**
  A text string that specifies the value of the runtime model attribute. Using the special value `*` is equivalent to not defining the attribute at all.

**Description**

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value `*`. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

**Note:** The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

**Example**

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

**See also**

`Checking module consistency`, page 73.

### __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.
Descriptions of pragma directives

Example

```c
#pragma __scanf_args
int printf(char const *,...);
/* Function call */
scanf("%d",x);  /* Compiler checks that x is an integer */
```

### 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`, `__segment_end`, and `__segment_size`. 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:

```c
void __memoryattribute *.
```

**Example**

```c
#pragma segment="MYSEGMENT" __data16 4
```

**See also**

*Dedicated segment operators*, page 109. For more information about segments and segment parts, see the chapter *Placing code and data*.

### STDC CX_LIMITED_RANGE

**Syntax**

```
#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}
```

**Parameters**

- **ON**: Normal complex mathematics formulas can be used.
- **OFF**: Normal complex mathematics formulas cannot be used.
- **DEFAULT**: Sets the default behavior, that is OFF.
Pragma directives

STDC FENV_ACCESS

Syntax

#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}

Parameters

ON Source code accesses the floating-point environment. Note that this argument is not supported by the compiler.
OFF Source code does not access the floating-point environment.
DEFAULT Sets the default behavior, that is OFF.

Description

Use this pragma directive to specify whether your source code accesses the floating-point environment or not.

Note: This directive is required by Standard C.

STDC FP_CONTRACT

Syntax

#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}

Parameters

ON The compiler is allowed to contract floating-point expressions.
OFF The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler.
DEFAULT Sets the default behavior, that is ON.

Description

Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C.

Example

#pragma STDC FP_CONTRACT=ON

Description

Use this pragma directive to specify that the compiler can use the normal complex mathematics formulas for \( \times \) (multiplication), \( / \) (division), and abs.

Note: This directive is required by Standard C. The directive is recognized but has no effect in the compiler.
**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 201.

Description

Use this pragma directive to specify IAR-specific *type attributes*, which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In this example, an `int` object with the memory attribute `__data16` is defined:

```
#pragma type_attribute=__data16
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__data16 int x;
```

See also

See the chapter *Extended keywords* for more details.

**vector**

Syntax

```
#pragma vector=vector1[, vector2, vector3, ...]
```

Parameters

```
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

```
#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><code>__bcd_add_type</code></td>
<td>Performs a binary coded decimal operation</td>
</tr>
<tr>
<td><code>__bic_SR_register</code></td>
<td>Clears bits in the SR register</td>
</tr>
<tr>
<td><code>__bic_SR_register_on_exit</code></td>
<td>Clears bits in the SR register when an interrupt or monitor function returns</td>
</tr>
<tr>
<td><code>__bis_SR_register</code></td>
<td>Sets bits in the SR register</td>
</tr>
<tr>
<td><code>__bis_SR_register_on_exit</code></td>
<td>Sets bits in the SR register when an interrupt or monitor function returns</td>
</tr>
<tr>
<td><code>__data16_read(addr)</code></td>
<td>Reads data to a 20-bit SFR register</td>
</tr>
<tr>
<td><code>__data16_write(addr)</code></td>
<td>Writes data to a 20-bit SFR register</td>
</tr>
<tr>
<td><code>__data20_read_type</code></td>
<td>Reads data which has a 20-bit address</td>
</tr>
<tr>
<td><code>__data20_write_type</code></td>
<td>Writes data which has a 20-bit address</td>
</tr>
<tr>
<td><code>__delay_cycles</code></td>
<td>Provides cycle-accurate delay functionality</td>
</tr>
<tr>
<td><code>__disable_interrupt</code></td>
<td>Disables interrupts</td>
</tr>
<tr>
<td><code>__enable_interrupt</code></td>
<td>Enables interrupts</td>
</tr>
</tbody>
</table>

Table 38: Intrinsic functions summary
Descriptions of intrinsic functions

<table>
<thead>
<tr>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__bcd_add_type</td>
<td>Makes switch statements rely on the specified value being even and within the specified range</td>
</tr>
<tr>
<td>__even_in_range</td>
<td>Makes switch statements rely on the specified value being even and within the specified range</td>
</tr>
<tr>
<td>__get_interrupt_state</td>
<td>Returns the interrupt state</td>
</tr>
<tr>
<td>__get_R4_register</td>
<td>Returns the value of the R4 register</td>
</tr>
<tr>
<td>__get_R5_register</td>
<td>Returns the value of the R5 register</td>
</tr>
<tr>
<td>__get_SP_register</td>
<td>Returns the value of the stack pointer</td>
</tr>
<tr>
<td>__get_SR_register</td>
<td>Returns the value of the SR register</td>
</tr>
<tr>
<td>__get_SR_register_on_exit</td>
<td>Returns the value that the processor status register will have when the current interrupt or monitor function returns</td>
</tr>
<tr>
<td>__low_power_mode_n</td>
<td>Enters a MSP430 low power mode</td>
</tr>
<tr>
<td>__low_power_mode_off_on_exit</td>
<td>Turns off low power mode when a monitor or interrupt function returns</td>
</tr>
<tr>
<td>__no_operation</td>
<td>Inserts a NOP instruction</td>
</tr>
<tr>
<td>__op_code</td>
<td>Inserts a constant into the instruction stream</td>
</tr>
<tr>
<td>__set_interrupt_state</td>
<td>Restores the interrupt state</td>
</tr>
<tr>
<td>__set_R4_register</td>
<td>Writes a specific value to the R4 register</td>
</tr>
<tr>
<td>__set_R5_register</td>
<td>Writes a specific value to the R5 register</td>
</tr>
<tr>
<td>__set_SP_register</td>
<td>Writes a specific value to the SP register</td>
</tr>
<tr>
<td>__swap_bytes</td>
<td>Executes the SWPB instruction</td>
</tr>
</tbody>
</table>

Table 38: Intrinsic functions summary (Continued)

__bcd_add_type

Syntax

```c
unsigned type __bcd_add_type(unsigned type x, unsigned type y);
```

where:

- **type** Can be one of the types short, long, or long long

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.
Intrinsic functions

Description
Performs a binary coded decimal addition. The parameters and the return value are represented as binary coded decimal (BCD) numbers, that is when a hexadecimal number (0x19) is used for representing the decimal number 19. The following functions are supported:

<table>
<thead>
<tr>
<th>Function</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>__bcd_add_short</td>
<td>Returns the sum of the two short parameters. The parameters and the return value are represented as four-digit BCD numbers.</td>
</tr>
<tr>
<td>__bcd_add_long</td>
<td>Returns the sum of the two long parameters. The parameters and the return value are represented as eight-digit BCD numbers.</td>
</tr>
<tr>
<td>__bcd_add_long_long</td>
<td>Returns the sum of the two long long parameters. The parameters and the return value are represented as sixteen-digit BCD numbers.</td>
</tr>
</tbody>
</table>

Table 39: Functions for binary coded decimal operations

Example
/* c = 0x19 */
c = __bcd_add_short(c, 0x01);
/* c = 0x20 */

__bic_SR_register
Syntax
void __bic_SR_register(unsigned short);
Description
Clears bits in the processor status register. The function takes an integer as its argument, that is, a bit mask with the bits to be cleared.

__bic_SR_register_on_exit
Syntax
void __bic_SR_register_on_exit(unsigned short);
Description
Clears bits in the processor status register when an interrupt or monitor function returns. The function takes an integer as its argument, that is, a bit mask with the bits to be cleared.

This intrinsic function is only available in interrupt and monitor functions.
__bis_SR_register
Syntax
void __bis_SR_register(unsigned short);
Description
Sets bits in the status register. The function takes an integer literal as its argument, that is, a bit mask with the bits to be set.

__bis_SR_register_on_exit
Syntax
void __bis_SR_register_on_exit(unsigned short);
Description
Sets bits in the processor status register when an interrupt or monitor function returns. The function takes an integer literal as its argument, that is, a bit mask with the bits to be set.
This intrinsic function is only available in interrupt and monitor functions.

__data16_read_addr
Syntax
unsigned long __data16_read_addr(unsigned short address);
where:

address Specifies the address for the read operation

Description
Reads data from a 20-bit SFR register located at the given 16-bit address. This intrinsic function is only useful on devices based on the MSP430X architecture.

Note: In the Small data model, and if interrupts are not configured to save all 20 bits of the registers they are using, interrupts must be disabled when you use this intrinsic function.

__data16_write_addr
Syntax
void __data16_write_addr(unsigned short address, unsigned long data);
where:

address Specifies the address for the write operation
data The data to be written
Part 2. Reference information

Intrinsic functions

__data20_read_type

**Description**

Writes a value to a 20-bit SFR register located at the given 16-bit address. This intrinsic function is only useful on devices based on the MSP430X architecture.

**Note:** In the Small data model, and if interrupts are not configured to save all 20 bits of the registers they are using, interrupts must be disabled when you use this intrinsic function.

**Syntax**

```c
unsigned type __data20_read_type(unsigned long address);
```

where:

- `address` Specifies the address for the read operation
- `type` Can be one of the types `char`, `short`, or `long`

**Description**

Reads data from the MSP430X full 1-Mbyte memory area. This intrinsic function is intended to be used in the Small data model. In the Medium and Large data models it is recommended to use `__data20` variables and pointers.

The following functions are supported:

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned char __data20_read_char</code></td>
<td>1 byte</td>
<td>1</td>
</tr>
<tr>
<td><code>unsigned short __data20_read_short</code></td>
<td>2 bytes</td>
<td>2</td>
</tr>
<tr>
<td><code>unsigned long __data20_read_long</code></td>
<td>4 bytes</td>
<td>2</td>
</tr>
</tbody>
</table>

*Table 40: Functions for reading data that has a 20-bit address*

**Note:** In the Small data model, and if interrupts are not configured to save all 20 bits of the registers they are using, interrupts must be disabled when you use this intrinsic function.

__data20_write_type

**Syntax**

```c
void __data20_write_type(unsigned long address, unsigned type);
```

where:

- `address` Specifies the address for the write operation
- `type` Can be one of the types `char`, `short`, or `long`
Descriptions of intrinsic functions

Description
Wrote data to the MSP430X full 1-Mbyte memory area. This intrinsic function is intended to be used in the Small data model. In the Medium and Large data models it is recommended to use __data20 variables and pointers.

The following functions are supported:

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char __data20_write_char</td>
<td>1 byte</td>
<td>1</td>
</tr>
<tr>
<td>unsigned short __data20_write_short</td>
<td>2 bytes</td>
<td>2</td>
</tr>
<tr>
<td>unsigned long __data20_write_long</td>
<td>4 bytes</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 41: Functions for writing data that has a 20-bit address

Note: In the Small data model, and if interrupts are not configured to save all 20 bits of the registers they are using, interrupts must be disabled when you use this intrinsic function.

__delay_cycles

Syntax
void __delay_cycles(unsigned long cycles);

Parameters
cycles The time delay in number of cycles. This must be a constant.

Description
Inserts assembler instructions that delay the execution the number of specified clock cycles, with a minimum of code.

__disable_interrupt

Syntax
void __disable_interrupt(void);

Description
Disables interrupts by inserting the DINT instruction.

__enable_interrupt

Syntax
void __enable_interrupt(void);

Description
Enables interrupts by inserting the EINT instruction.
__even_in_range

Syntax
unsigned short __even_in_range(unsigned short value,
                unsigned short upper_limit);

Parameters
value The switch expression
upper_limit The last value in the allowed range

Description
Instructs the compiler to rely on the specified value being even and within the specified
range. The code will be generated accordingly and will only work if the requirement is
fulfilled.

This intrinsic function can be used for achieving optimal code for switch statements
where you know that the only values possible are even values within a given range, for
example an interrupt service routine for an Interrupt Vector Generator interrupt.

Example
switch (__even_in_range(TAIV, 10))

See also
Interrupt Vector Generator interrupt functions, page 25.

__get_interrupt_state

Syntax
__istate_t __get_interrupt_state(void);

Description
Returns the global interrupt state. The return value can be used as an argument to the
__set_interrupt_state intrinsic function, which will restore the interrupt state.

Example
__istate_t s = __get_interrupt_state();
__disable_interrupt();
/* Do something here. */
__set_interrupt_state(s);

The advantage of using this sequence of code compared to using
__disable_interrupt and __enable_interrupt is that the code in this example
will not enable any interrupts disabled before the call of __get_interrupt_state.
Descriptions of intrinsic functions

__get_R4_register
Syntax: unsigned short __get_R4_register(void);
Description: Returns the value of the R4 register. This intrinsic function is only available when the register is locked.
See also: --lock_r4, page 171

__get_R5_register
Syntax: unsigned short __get_R5_register(void);
Description: Returns the value of the R5 register. This intrinsic function is only available when the register is locked.
See also: --lock_r5, page 171

__get_SP_register
Syntax: unsigned short __get_SP_register(void);
Description: Returns the value of the stack pointer register SP.

__get_SR_register
Syntax: unsigned short __get_SR_register(void);
Description: Returns the value of the processor status register SR.

__get_SR_register_on_exit
Syntax: unsigned short __get_SR_register_on_exit(void);
Description: Returns the value that the processor status register SR will have when the current interrupt or monitor function returns.
This intrinsic function is only available in interrupt and monitor functions.
### __low_power_mode_n

**Syntax**

```c
void __low_power_mode_n(void);
```

**Description**

Enters a MSP430 low power mode, where `n` can be one of 0–4. This also enables global interrupts by setting the GIE bit in the status register.

### __low_power_mode_off_on_exit

**Syntax**

```c
void __low_power_mode_off_on_exit(void);
```

**Description**

Turns off the low power mode when a monitor or interrupt function returns. This intrinsic function is only available in interrupt and monitor functions.

### __no_operation

**Syntax**

```c
void __no_operation(void);
```

**Description**

Inserts a `NOP` instruction.

### __op_code

**Syntax**

```c
void __op_code(unsigned short);
```

**Description**

Emits the 16-bit value into the instruction stream for the current function by inserting a DC16 constant.

### __set_interrupt_state

**Syntax**

```c
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 237.
Descriptions of intrinsic functions

__set_R4_register
Syntax
void __set_R4_register(unsigned short);
Description
Writes a specific value to the R4 register. This intrinsic function is only available when R4 is locked.
See also
--lock_r4, page 171

__set_R5_register
Syntax
void __set_R5_register(unsigned short);
Description
Writes a specific value to the R5 register. This intrinsic function is only available when R5 is locked.
See also
--lock_r5, page 171

__set_SP_register
Syntax
void __set_SP_register(unsigned short);
Description
Writes a specific value to the SP stack pointer register. A warning message is issued if the compiler has used the stack in any way at the location where this intrinsic function is used.

__swap_bytes
Syntax
unsigned short __swap_bytes(unsigned short);
Description
Inserts an SWPB instruction and returns the argument with the upper and lower parts interchanged.
Example
__swap_bytes(0x1234)
returns 0x3412.
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 MSP430 adheres to Standard C. The compiler also makes these preprocessor-related features available to you:

- Predefined preprocessor symbols
  These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see Descriptions of predefined preprocessor symbols, page 242.

- 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 160.

- Preprocessor extensions
  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapterPragma 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 244.

- Preprocessor output
  Use the option --preprocess to direct preprocessor output to a named file, see --preprocess, page 180.

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

#include "mydirectory/myfile"

In source code, use forward slashes:

file = fopen("mydirectory/myfile","rt");

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.
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 243, and --no_path_in_file_macros, page 175.</td>
</tr>
<tr>
<td><strong>BUILD_NUMBER</strong></td>
<td>A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.</td>
</tr>
<tr>
<td><strong>CORE</strong></td>
<td>An integer that identifies the chip core in use. The symbol reflects the --core option and is defined to <strong>430</strong> for the MSP430 architecture and to <strong>430X</strong> for the MSP430X architecture. These symbolic names can be used when testing the <strong>CORE</strong> symbol.</td>
</tr>
<tr>
<td>__cplusplus</td>
<td>An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with #ifdef to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*</td>
</tr>
<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 is defined to <strong>DATA_MODEL_SMALL</strong>, <strong>DATA_MODEL_MEDIUM</strong>, or <strong>DATA_MODEL_LARGE</strong>.</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 2010&quot;.</td>
</tr>
<tr>
<td><strong>DOUBLE</strong></td>
<td>An integer that identifies the size of the data type double. The symbol reflects the --double option and is defined to 32 or 64.</td>
</tr>
</tbody>
</table>

Table 42: Predefined symbols
<table>
<thead>
<tr>
<th>Predefined symbol</th>
<th>Identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__embedded_cplusplus</code></td>
<td>An integer which is defined to 1 when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*</td>
</tr>
<tr>
<td><code>__FILE__</code></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 <code>__BASE_FILE__</code>, page 242, and <code>--no_path_in_file_macros</code>, page 175.*</td>
</tr>
<tr>
<td><code>__func__</code></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 <code>-e</code>, page 167. See also <code>__PRETTY_FUNCTION__</code>, page 243.</td>
</tr>
<tr>
<td><code>__FUNCTION__</code></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 <code>-e</code>, page 167. See also <code>__PRETTY_FUNCTION__</code>, page 243.</td>
</tr>
<tr>
<td><code>__IAR_SYSTEMS_ICC__</code></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 <code>#ifdef</code> to detect whether the code was compiled by a compiler from IAR Systems.</td>
</tr>
<tr>
<td><code>__ICC430__</code></td>
<td>An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for MSP430.</td>
</tr>
<tr>
<td><code>__LINE__</code></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><code>__POSITION_INDEPENDENT_CODE__</code></td>
<td>An integer that is set to 1 when the code is compiled with the option <code>--pic</code>.</td>
</tr>
<tr>
<td><code>__PRETTY_FUNCTION__</code></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 <code>void func(char)</code>. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code>, page 167. See also <code>__func__</code>, page 243.*</td>
</tr>
</tbody>
</table>

Table 42: 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 Standard C directives.

### NDEBUG

**Description**

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

---

<table>
<thead>
<tr>
<th>Predefined symbol</th>
<th>Identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REGISTER_MODEL</strong></td>
<td>An integer that equals one of the following:</td>
</tr>
<tr>
<td></td>
<td><strong>REGISTER_MODEL_REG16</strong> (for 430 and 430X in the Small data model) or</td>
</tr>
<tr>
<td></td>
<td><strong>REGISTER_MODEL_REG20</strong> (for 430X in the Medium and Large data model).</td>
</tr>
</tbody>
</table>
| __STDC__                  | An integer that is set to 1, which means the compiler adheres to Standard C. This symbol can be tested with 
                           | #ifdef to detect whether the compiler in use adheres to Standard C.*      |
| __STDC_VERSION__          | An integer that identifies the version of the C standard in use. The symbol expands to 199901L unless the --c89 compiler option is used in which case the symbol expands to 199409L. This symbol does not apply in EC++ mode.* |
| __SUBVERSION__            | An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4. |
| __TIME__                  | A string that identifies the time of compilation in the form "hh:mm:ss".*  |
| __VER__                   | An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: (100 * the major version number + the minor version number). For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of __VER__ is 334. |

* This symbol is required by Standard C.

---
If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

See also

*Assert*, page 72.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

### #warning message

**Syntax**

```
#define warning_message
```

where `message` can be any string.

**Description**

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.
Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

Library overview

The compiler provides two different libraries:

- IAR DLIB Library is a complete library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.
- IAR CLIB Library is a light-weight library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support C++. Note that the legacy CLIB library is provided for backward compatibility and should not be used for new application projects.

Note that different customization methods are normally needed for these two libraries. For additional information, see the chapter "The DLIB runtime environment" and "The CLIB runtime environment," respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter "Implementation-defined behavior" in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the \#include directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.
LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from
the library object files that are supplied with the product. For information about how to
choose a runtime library, see Basic project configuration, page 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 the following functions and parts are
not reentrant because they need static data:

- Heap functions—malloc, free, realloc, calloc, and the C++ operators new and delete
- Time functions—asctime, localtime, gmtime, mktime
- Multi-byte functions—mbrlen, mbstowc, mbstowc, wcrtomb, wcsrtomb, wctomb
- The miscellaneous functions setlocale, rand, atexit, strerror, strtok
- Functions that use files or the heap in some way. This includes printf, sprintf, scanf, scanf, getchar, and putchar.

For the CLIB library, the qsort function and functions that use files in some way are
non-reentrant. This includes printf, scanf, getchar, and putchar. However, the
functions sprintf and sscanf are reentrant.

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, strtof, strto1, stroul

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 Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter Implementation-defined behavior in this guide.
- Standard C library definitions, for user programs.
- 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 MSP430 features. See the chapter Intrinsic functions for more information.

In addition, the IAR DLIB Library includes some added C functionality, see Added C functionality, page 253.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter Using C.

This table lists the C header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert.h</td>
<td>Enforcing assertions when functions execute</td>
</tr>
<tr>
<td>complex.h</td>
<td>Computing common complex mathematical functions</td>
</tr>
<tr>
<td>ctype.h</td>
<td>Classifying characters</td>
</tr>
<tr>
<td>errno.h</td>
<td>Testing error codes reported by library functions</td>
</tr>
<tr>
<td>fenv.h</td>
<td>Floating-point exception flags</td>
</tr>
<tr>
<td>float.h</td>
<td>Testing floating-point type properties</td>
</tr>
<tr>
<td>inttypes.h</td>
<td>Defining formatters for all types defined in stdint.h</td>
</tr>
<tr>
<td>iso646.h</td>
<td>Using Amendment 1—iso646.h standard header</td>
</tr>
<tr>
<td>limits.h</td>
<td>Testing integer type properties</td>
</tr>
<tr>
<td>locale.h</td>
<td>Adapting to different cultural conventions</td>
</tr>
</tbody>
</table>

Table 43: Traditional Standard C header files—DLIB
C++ HEADER FILES

This section lists the C++ header files.

**Embedded C++**

This table lists the Embedded C++ header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex</td>
<td>Defining a class that supports complex arithmetic</td>
</tr>
<tr>
<td>exception</td>
<td>Defining several functions that control exception handling</td>
</tr>
<tr>
<td>fstream</td>
<td>Defining several I/O stream classes that manipulate external files</td>
</tr>
<tr>
<td>iomanip</td>
<td>Declaring several I/O stream manipulators that take an argument</td>
</tr>
<tr>
<td>ios</td>
<td>Defining the class that serves as the base for many I/O streams classes</td>
</tr>
<tr>
<td>iosfwd</td>
<td>Declaring several I/O stream classes before they are necessarily defined</td>
</tr>
<tr>
<td>istream</td>
<td>Declaring the I/O stream objects that manipulate the standard streams</td>
</tr>
<tr>
<td>new</td>
<td>Defining the class that performs extractions</td>
</tr>
<tr>
<td>wchar</td>
<td>Supporting wide characters</td>
</tr>
<tr>
<td>wctype</td>
<td>Classifying wide characters</td>
</tr>
</tbody>
</table>

Table 44: Embedded C++ header files
## 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 45: Standard template library header files
### Using Standard C libraries in C++

The C++ library works in conjunction with some of the header files from the Standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

This table shows the new header files:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cassert</code></td>
<td>Enforcing assertions when functions execute</td>
</tr>
<tr>
<td><code>ccomplex</code></td>
<td>Computing common complex mathematical functions</td>
</tr>
<tr>
<td><code>cctype</code></td>
<td>Classifying characters</td>
</tr>
<tr>
<td><code>cerrno</code></td>
<td>Testing error codes reported by library functions</td>
</tr>
<tr>
<td><code>cfenv.h</code></td>
<td>Floating-point exception flags</td>
</tr>
<tr>
<td><code>cfloat</code></td>
<td>Testing floating-point type properties</td>
</tr>
<tr>
<td><code>cinttypes</code></td>
<td>Defining formatters for all types defined in <code>stdint.h</code></td>
</tr>
<tr>
<td><code>ciso646</code></td>
<td>Using Amendment 1—<code>iso646.h</code> standard header</td>
</tr>
<tr>
<td><code>climits</code></td>
<td>Testing integer type properties</td>
</tr>
<tr>
<td><code>clocale</code></td>
<td>Adapting to different cultural conventions</td>
</tr>
<tr>
<td><code>cmath</code></td>
<td>Computing common mathematical functions</td>
</tr>
<tr>
<td><code>csetjmp</code></td>
<td>Executing non-local goto statements</td>
</tr>
<tr>
<td><code>csignal</code></td>
<td>Controlling various exceptional conditions</td>
</tr>
<tr>
<td><code>cstdarg</code></td>
<td>Accessing a varying number of arguments</td>
</tr>
<tr>
<td><code>cstdbool</code></td>
<td>Adds support for the <code>bool</code> data type in C.</td>
</tr>
<tr>
<td><code>cstddef</code></td>
<td>Defining several useful types and macros</td>
</tr>
<tr>
<td><code>cstdint</code></td>
<td>Providing integer characteristics</td>
</tr>
<tr>
<td><code>cstdio</code></td>
<td>Performing input and output</td>
</tr>
<tr>
<td><code>cstdlib</code></td>
<td>Performing a variety of operations</td>
</tr>
<tr>
<td><code>cstring</code></td>
<td>Manipulating several kinds of strings</td>
</tr>
<tr>
<td><code>ctgmath.h</code></td>
<td>Type-generic mathematical functions</td>
</tr>
<tr>
<td><code>ctime</code></td>
<td>Converting between various time and date formats</td>
</tr>
<tr>
<td><code>cwchar</code></td>
<td>Support for wide characters</td>
</tr>
<tr>
<td><code>cwctype</code></td>
<td>Classifying wide characters</td>
</tr>
</tbody>
</table>

Table 46: 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.

The following include files provide these features:
- `fenv.h`
- `stdio.h`
- `stdlib.h`
- `string.h`

**fenv.h**

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapenable` and `fegetrapdisable`. No floating-point status flags are supported.

**stdio.h**

These functions provide additional I/O functionality:

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

These are the additional functions defined in string.h:

- **strdup**: Duplicates a string on the heap.
- **strcasecmp**: Compares strings case-insensitive.
- **strncasecmp**: Compares strings case-insensitive and bounded.
- **strnlen**: Bounded string length.

**IAR CLIB Library**

The IAR CLIB Library provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- **Standard C library definitions available for user programs.** These are documented in this chapter.
- **The system startup code.** It is described in the chapter *The CLIB runtime environment* in this guide.
- **Runtime support libraries; for example low-level floating-point routines.**
- **Intrinsic functions, allowing low-level use of MSP430 features.** See the chapter *Intrinsic functions* for more information.

**LIBRARY DEFINITIONS SUMMARY**

This following table lists the header files specific to the CLIB library:

<table>
<thead>
<tr>
<th>Header file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert.h</td>
<td>Assertions</td>
</tr>
<tr>
<td>ctype.h*</td>
<td>Character handling</td>
</tr>
<tr>
<td>errno.h</td>
<td>Error return values</td>
</tr>
<tr>
<td>float.h</td>
<td>Limits and sizes of floating-point types</td>
</tr>
<tr>
<td>iccbutl.h</td>
<td>Low-level routines</td>
</tr>
<tr>
<td>limits.h</td>
<td>Limits and sizes of integral types</td>
</tr>
<tr>
<td>math.h</td>
<td>Mathematics</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>Non-local jumps</td>
</tr>
<tr>
<td>stdarg.h</td>
<td>Variable arguments</td>
</tr>
<tr>
<td>stdbool.h</td>
<td>Adds support for the bool data type in C</td>
</tr>
</tbody>
</table>

*Table 47: IAR CLIB Library header files*
<table>
<thead>
<tr>
<th>Header file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stddef.h</td>
<td>Common definitions including size_t, NULL, ptrdiff_t, and offsetof</td>
</tr>
<tr>
<td>stdio.h</td>
<td>Input/output</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>General utilities</td>
</tr>
<tr>
<td>string.h</td>
<td>String handling</td>
</tr>
</tbody>
</table>

Table 47: IAR CLIB Library header files (Continued)

* The functions isxxx, toupper, and tolower declared in the header file ctype.h evaluate their argument more than once. This is not according to the ISO/ANSI standard.
Segment reference

The compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For more information about segments, see the chapter *Placing code and data*.

Summary of segments

The table below lists the segments that are available in the compiler:

<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>CODE</td>
<td>Holds program code.</td>
</tr>
<tr>
<td>CODE_I</td>
<td>Holds code declared __ramfunc.</td>
</tr>
<tr>
<td>CODE_ID</td>
<td>Holds code copied to CODE_I at startup.</td>
</tr>
<tr>
<td>CSTACK</td>
<td>Holds the stack used by C or C++ programs.</td>
</tr>
<tr>
<td>CSTART</td>
<td>Holds the startup code.</td>
</tr>
<tr>
<td>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_HEAP</td>
<td>Holds the heap used for dynamically allocated data in data16 memory.</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_P</td>
<td>Holds __data16 variables defined with the __persistent keyword.</td>
</tr>
<tr>
<td>DATA16_Z</td>
<td>Holds zero-initialized __data16 static and global variables.</td>
</tr>
<tr>
<td>DATA20_AC</td>
<td>Holds __data20 located constant data.</td>
</tr>
<tr>
<td>DATA20_AN</td>
<td>Holds __data20 located uninitialized data.</td>
</tr>
<tr>
<td>DATA20_C</td>
<td>Holds __data20 constant data.</td>
</tr>
</tbody>
</table>

Table 48: Segment summary
### Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives `-Z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—`CODE`, `CONST`, or `DATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 5, `XLINK` segment memory types, page 32.

For information about the `-Z` and the `-P` directives, see the *IAR Linker and Library Tools Reference Guide*.

For information about how to define segments in the linker configuration file, see *Customizing the linker configuration file*, page 33.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA20_HEAP</td>
<td>Holds the heap used for dynamically allocated data in data20 memory.</td>
</tr>
<tr>
<td>DATA20_I</td>
<td>Holds <code>__data20</code> static and global initialized variables.</td>
</tr>
<tr>
<td>DATA20_ID</td>
<td>Holds initial values for <code>__data20</code> static and global variables in DATA20_I.</td>
</tr>
<tr>
<td>DATA20_N</td>
<td>Holds <code>__no_init__data20</code> static and global variables.</td>
</tr>
<tr>
<td>DATA20_P</td>
<td>Holds <code>__data20</code> variables defined with the <code>__persistent</code> keyword.</td>
</tr>
<tr>
<td>DATA20_Z</td>
<td>Holds zero-initialized <code>__data20</code> static and global variables.</td>
</tr>
<tr>
<td>DIFUNCT</td>
<td>Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before <code>main</code> is called.</td>
</tr>
<tr>
<td>INFO</td>
<td>Holds data to be placed in the MSP430 information memory.</td>
</tr>
<tr>
<td>INFOA</td>
<td>Holds data to be placed in bank A of the MSP430 information memory.</td>
</tr>
<tr>
<td>INFOB</td>
<td>Holds data to be placed in bank B of the MSP430 information memory.</td>
</tr>
<tr>
<td>INFOC</td>
<td>Holds data to be placed in bank C of the MSP430 information memory.</td>
</tr>
<tr>
<td>INFOD</td>
<td>Holds data to be placed in bank D of the MSP430 information memory.</td>
</tr>
<tr>
<td>INTVEC</td>
<td>Contains the reset and interrupt vectors.</td>
</tr>
<tr>
<td>ISR_CODE</td>
<td>Holds interrupt functions when compiling for the MSP430X architecture.</td>
</tr>
<tr>
<td>REGVAR_AN</td>
<td>Holds <code>__regvar</code> data.</td>
</tr>
<tr>
<td>RESET</td>
<td>Holds the reset vector.</td>
</tr>
</tbody>
</table>

Table 48: Segment summary (Continued)
For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

**CHECKSUM**

Description: Holds the checksum bytes generated by the linker. This segment also holds the `__checksum` symbol. Note that the size of this segment is affected by the linker option `-J`.

Segment memory type: CONST

Memory placement: This segment can be placed anywhere in ROM memory.

Access type: Read-only

**CODE**

Description: Holds program code, except the code for system initialization.

Segment memory type: CODE

Memory placement:
- MSP430: 0x0002–0xFFF
- MSP430X: 0x00002–0xFFFFE

Access type: Read-only

**CODE_I**

Description: Holds program code declared `__ramfunc`. This code will be executed in RAM. The code is copied from `CODE_ID` during initialization.

Segment memory type: DATA

Memory placement:
- MSP430: 0x0002–0xFFF
- MSP430X: 0x00002–0xFFFFE

Access type: Read/write
<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
<th>Segment memory type</th>
<th>Memory placement</th>
<th>Access type</th>
<th>See also</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CODE_ID</strong></td>
<td>This is the permanent storage of program code declared __ramfunc. This code will be executed in RAM. The code is copied to CODE_I during initialization.</td>
<td>CODE</td>
<td>MSP430: 0x0002–0xFFFD MSP430X: 0x00002–0xFFFFE</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>CSTACK</strong></td>
<td>Holds the internal data stack.</td>
<td>DATA</td>
<td>0x0002–0xFFFD (also for the MSP430X architecture)</td>
<td>Read/write</td>
<td><em>The stack</em>, page 38.</td>
</tr>
<tr>
<td><strong>CSTART</strong></td>
<td>Holds the startup code. This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.</td>
<td>CODE</td>
<td>0x0002–0xFFFD</td>
<td>Read-only</td>
<td></td>
</tr>
</tbody>
</table>
### DATA16_AC

**Description**
Holds __data16 located constant data.  

*Located* means being placed at an absolute location using the @ operator or the #pragma location directive. Because the location is known, this segment does not need to be specified in the linker configuration file.

### DATA16_AN

**Description**
Holds __no_init __data16 located data.  

*Located* means being placed at an absolute location using the @ operator or the #pragma location directive. Because the location is known, this segment does not need to be specified in the linker configuration file.

### DATA16_C

**Description**
Holds __data16 constant data. This can include constant variables, string and aggregate literals, etc.

**Segment memory type**
CONST

**Memory placement**
0x0001–0xFFFD

**Access type**
Read-only

### DATA16_HEAP

**Description**
Holds the heap used for dynamically allocated data in data16 memory, for example data allocated by malloc and free, and in C++ new and delete.

**Segment memory type**
DATA

**Memory placement**
0x0002–0xFFFD

**Access type**
Read/write

**See also**
The heap, page 39 and New and Delete operators, page 119.
Descriptions of segments

**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 configuration file, the -Z directive must be used.

**Segment memory type** DATA
**Memory placement** 0x0001–0xFFFD
**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 configuration file, the -Z directive must be used.

**Segment memory type** CONST
**Memory placement** 0x0001–0xFFFD
**Access type** Read-only

**DATA16_N**

**Description**
Holds static and global __no_init __data16 variables.

**Segment memory type** DATA
**Memory placement** 0x0001–0xFFFD
**Access type** Read/write
### DATA16_P

**Description**
Holds static and global `__data16` variables defined with the `__persistent` keyword. These variables will only be initialized, for example, by a code downloader, and not by `cstart`.

**Segment memory type**
DATA

**Memory placement**
0x0001–0xFFFD

**Access type**
Read/write

**See also**
`__persistent`, page 208.

### DATA16_Z

**Description**
Holds zero-initialized `__data16` static and global variables. The contents of this segment is declared by the system startup code.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

**Segment memory type**
DATA

**Memory placement**
0x0001–0xFFFD

**Access type**
Read/write

### DATA20_AC

**Description**
Holds `__data20` located constant data.

*Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker configuration file.
Descriptions of segments

**DATA20_AN**

Description: Holds __no_init__ __data20__ located data.

Located means being placed at an absolute location using the @ operator or the #pragma location directive. Because the location is known, this segment does not need to be specified in the linker configuration file.

**DATA20_C**

Description: Holds __data20__ constant data. This can include constant variables, string and aggregate literals, etc.

Segment memory type: CONST
Memory placement: 0x00001–0xFFFFE
Access type: Read-only

**DATA20_HEAP**

Description: Holds the heap used for dynamically allocated data in __data20__ memory, for example data allocated by __data20_malloc__ and __data20_free__, and in C++, new and delete.

Segment memory type: DATA
Memory placement: 0x00001–0xFFFFE
Access type: Read/write
See also: *The heap*, page 39 and *New and Delete operators*, page 119.

**DATA20_I**

Description: Holds __data20__ static and global initialized variables initialized by copying from the segment DATA20_ID at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type: DATA
<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
<th>Memory placement</th>
<th>Access type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATA20_ID</strong></td>
<td>Holds initial values for __data20 static and global variables in the DATA20_I segment. These values are copied from DATA20_ID to DATA20_I at application startup. This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.</td>
<td>0x00001–0xFFFFE</td>
<td>Read/write</td>
</tr>
<tr>
<td><strong>DATA20_N</strong></td>
<td>Holds static and global __no_init __data20 variables.</td>
<td>0x00001–0xFFFFE</td>
<td>Read-only</td>
</tr>
<tr>
<td><strong>DATA20_P</strong></td>
<td>Holds static and global __data20 variables defined with the __persistent keyword. These variables will only be initialized, for example, by a code downloader, and not by cstartup.</td>
<td>0x00001–0xFFFFE</td>
<td>Read/write</td>
</tr>
</tbody>
</table>
### Descriptions of segments

See also **__persistent**, page 208.

<table>
<thead>
<tr>
<th><strong>DATA20_Z</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Holds zero-initialized <code>__data20</code> static and global variables. The contents of this segment is declared by the system startup code. This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</td>
</tr>
<tr>
<td><strong>Segment memory type</strong></td>
<td>DATA</td>
</tr>
<tr>
<td><strong>Memory placement</strong></td>
<td>0x00001–0xFFFFE</td>
</tr>
<tr>
<td><strong>Access type</strong></td>
<td>Read/write</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>DIFUNCT</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Holds the dynamic initialization vector used by C++.</td>
</tr>
<tr>
<td><strong>Segment memory type</strong></td>
<td>CONST</td>
</tr>
<tr>
<td><strong>Memory placement</strong></td>
<td>This segment must be placed in the first 64 Kbytes of memory.</td>
</tr>
<tr>
<td><strong>Access type</strong></td>
<td>Read-only</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>INFO</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Holds data to be placed in the MSP430 information memory. Note that the INFO segment and the INFOA–INFOD segments overlap.</td>
</tr>
<tr>
<td><strong>Segment memory type</strong></td>
<td>CONST</td>
</tr>
<tr>
<td><strong>Memory placement</strong></td>
<td>Depends on the device.</td>
</tr>
<tr>
<td><strong>Access type</strong></td>
<td>Read-only</td>
</tr>
</tbody>
</table>
INFOA

Description: Holds data to be placed in bank A of the MSP430 information memory. Note that the INFOA segment and the INFO segment overlap.

Segment memory type: CONST

Memory placement: Depends on the device.

Access type: Read-only

INFOB

Description: Holds data to be placed in bank B of the MSP430 information memory. Note that the INFOB segment and the INFO segment overlap.

Segment memory type: CONST

Memory placement: Depends on the device.

Access type: Read-only

INFOC

Description: Holds data to be placed in bank C of the MSP430 information memory. Note that the INFOC segment and the INFO segment overlap.

Segment memory type: CONST

Memory placement: Depends on the device.

Access type: Read-only

INFOD

Description: Holds data to be placed in bank D of the MSP430 information memory. Note that the INFOD segment and the INFO segment overlap.

Segment memory type: CONST

Memory placement: Depends on the device.
Descriptions of segments

**Access type**  Read-only

**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  CODE

Memory placement  This segment can be placed anywhere in memory.

Access type  Read-only

**ISR_CODE**

Description  Holds interrupt functions when compiling for the MSP430X architecture. This segment is not used when compiling for the MSP430 architecture.

Segment memory type  CODE

Memory placement  0x0002–0xFFFF

Access type  Read-only

**REGVAR_AN**

Description  Holds `__regvar` data.

Memory placement  This segment is placed in the memory area reserved for registers, and does not occupy space in the normal memory area.

**RESET**

Description  Holds the reset vector.

Segment memory type  CODE

Memory placement  0xFFFFE–0xFFFFF

Access type  Read-only
Implementation-defined behavior

This chapter describes how IAR Systems handles the implementation-defined areas of the C language.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

The text in this chapter applies to the DLIB library. Because the CLIB library does not follow Standard C, its implementation-defined behavior is not documented. For information about the CLIB library, see The CLIB runtime environment, page 77.

Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

J.3.1 Translation

Diagnostics (3.10, 5.1.1.3)

Diagnostics are produced in the form:

filename,linenumber level[tag]: message

where filename is the name of the source file in which the error was encountered, linenumber is the line number at which the compiler detected the error, level is the level of seriousness of the message (remark, warning, error, or fatal error), tag is a unique tag that identifies the message, and message is an explanatory message, possibly several lines.

White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.
J3.2 Environment

The character set (5.1.1.2)
The source character set is the same as the physical source file multibyte character set.

The source character set is the same as the physical source file multibyte character set.

Descriptions of implementation-defined behavior

Main (5.1.2.1)
The function called at program startup is called main. No prototype is declared for main, and the only definition supported for main is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see Customizing system initialization, page 61.

The effect of program termination (5.1.2.1)
Terminating the application returns the execution to the startup code (just after the call to main).

Alternative ways to define main (5.1.2.1)
There is no alternative ways to define the main function.

The argv argument (5.1.2.1)
The argv argument is not supported.

Streams as interactive devices (5.1.2.3)
The streams stdin, stdout, and stderr are treated as interactive devices.

Signals, their semantics, and the default handling (7.14)
In the DLIB library, the set of supported signals is the same as in Standard C. A raised signal will do nothing; unless the signal function is customized to fit the application.

Signal values for computational exceptions (7.14.1.1)
In the DLIB library, there are no implementation-defined values that correspond to a computational exception.
Signals at system startup (7.14.1.1)
In the DLIB library, there are no implementation-defined signals that are executed at system startup.

Environment names (7.20.4.5)
In the DLIB library, there are no implementation-defined environment names that are used by the `getenv` function.

The system function (7.20.4.6)
The `system` function is not supported.

J.3.3 Identifiers

Multibyte characters in identifiers (6.4.2)
Additional multibyte characters may not appear in identifiers.

Significant characters in identifiers (5.2.4.1, 6.1.2)
The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

J.3.4 Characters

Number of bits in a byte (3.6)
A byte contains 8 bits.

Execution character set member values (5.2.1)
The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the host character set.

Alphabetic escape sequences (5.2.2)
The standard alphabetic escape sequences have the values \a–7, \b–8, \f–12, \n–10, \r–13, \t–9, and \v–11.

Characters outside of the basic executive character set (6.2.5)
A character outside of the basic executive character set that is stored in a `char` is not transformed.
Descriptions of implementation-defined behavior

Plain char (6.2.5, 6.3.1.1)
A plain char is treated as an unsigned char.

Source and execution character sets (6.4.4.4, 5.1.1.2)
The source character set is the set of legal characters that can appear in source files. By default, the source character set is the standard ASCII character set. However, if you use the command line option --enable_multibytes, the source character set will be the host computer’s default character set.

The execution character set is the set of legal characters that can appear in the execution environment. By default, the execution character set is the standard ASCII character set. However, if you use the command line option --enable_multibytes, the execution character set will be the host computer’s default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. See Locale, page 67.

Integer character constants with more than one character (6.4.4.4)
An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

Wide character constants with more than one character (6.4.4.4)
A wide character constant that contains more than one multibyte character generates a diagnostic message.

Locale used for wide character constants (6.4.4.4)
By default, the C locale is used. If the --enable_multibytes compiler option is used, the default host locale is used instead.

Locale used for wide string literals (6.4.5)
By default, the C locale is used. If the --enable_multibytes compiler option is used, the default host locale is used instead.

Source characters as executive characters (6.4.5)
All source characters can be represented as executive characters.
J.3.5 Integers

Extended integer types (6.2.5)
There are no extended integer types.

Range of integer values (6.2.6.2)
The representation of integer values are in the two’s complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.
See Basic data types, page 190, for information about the ranges for the different integer types.

The rank of extended integer types (6.3.1.1)
There are no extended integer types.

Signals when converting to a signed integer type (6.3.1.3)
No signal is raised when an integer is converted to a signed integer type.

Signed bitwise operations (6.5)
Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

J.3.6 Floating point

Accuracy of floating-point operations (5.2.4.2.2)
The accuracy of floating-point operations is unknown.

Rounding behaviors (5.2.4.2.2)
There are no non-standard values of FLT_ROUNDS.

Evaluation methods (5.2.4.2.2)
There are no non-standard values of FLT_EVAL_METHOD.

Converting integer values to floating-point values (6.3.1.4)
When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (FLT_ROUNDS is defined to 1).
Converting floating-point values to floating-point values (6.3.1.5)
When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (FLT_ROUNDS is defined to 1).

Denoting the value of floating-point constants (6.4.4.2)
The round-to-nearest rounding mode is used (FLT_ROUNDS is defined to 1).

Contraction of floating-point values (6.5)
Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

Default state of FENV_ACCESS (7.6.1)
The default state of the pragma directive FENV_ACCESS is OFF.

Additional floating-point mechanisms (7.6, 7.12)
There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

Default state of FP_CONTRACT (7.12.2)
The default state of the pragma directive FP_CONTRACT is OFF.

J.3.7 Arrays and pointers

Conversion from/to pointers (6.3.2.3)
See Casting, page 195, for information about casting of data pointers and function pointers.

ptrdiff_t (6.5.6)
For information about ptrdiff_t, see ptrdiff_t, page 196.

J.3.8 Hints

Honoring the register keyword (6.7.1)
User requests for register variables are not honored.
Inlining functions (6.7.4)
User requests for inlining functions increases the chance, but does not make it certain, that the function will actually be inlined into another function. See the pragma directive `inline`, page 220.

J.3.9 Structures, unions, enumerations, and bitfields

Sign of 'plain' bitfields (6.7.2, 6.7.2.1)
For information about how a 'plain' int bitfield is treated, see Bitfields, page 191.

Possible types for bitfields (6.7.2.1)
All integer types can be used as bitfields in the compiler’s extended mode, see `-e`, page 167.

Bitfields straddling a storage-unit boundary (6.7.2.1)
A bitfield is always placed in one—and one only—storage unit, which means that the bitfield cannot straddle a storage-unit boundary.

Allocation order of bitfields within a unit (6.7.2.1)
For information about how bitfields are allocated within a storage unit, see Bitfields, page 191.

Alignment of non-bitfield structure members (6.7.2.1)
The alignment of non-bitfield members of structures is the same as for the member types, see Alignment, page 189.

Integer type used for representing enumeration types (6.7.2.2)
The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

J.3.10 Qualifiers

Access to volatile objects (6.7.3)
Any reference to an object with volatile qualified type is an access, see Declaring objects volatile, page 199.
J.3.11 Preprocessing directives

Mapping of header names (6.4.7)
Sequences in header names are mapped to source file names verbatim. A backslash '\' is not treated as an escape sequence. See Overview of the preprocessor, page 241.

Character constants in constant expressions (6.10.1)
A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

The value of a single-character constant (6.10.1)
A single-character constant may only have a negative value if a plain character (char) is treated as a signed character, see --char_is_signed, page 159.

Including bracketed filenames (6.10.2)
For information about the search algorithm used for file specifications in angle brackets <>, see Include file search procedure, page 148.

Including quoted filenames (6.10.2)
For information about the search algorithm used for file specifications enclosed in quotes, see Include file search procedure, page 148.

Preprocessing tokens in #include directives (6.10.2)
Preprocessing tokens in an #include directive are combined in the same way as outside an #include directive.

Nesting limits for #include directives (6.10.2)
There is no explicit nesting limit for #include processing.

Universal character names (6.10.3.2)
Universal character names (UCN) are not supported.

Recognized pragma directives (6.10.6)
In addition to the pragma directives described in the chapterPragma directives, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the Pragma directives chapter and here, the information provided in the Pragma directives chapter overrides the information here.

alignment
Default *__DATE__* and *__TIME__* (6.10.8)
The definitions for *__TIME__* and *__DATE__* are always available.

**J.3.12 Library functions**

**Additional library facilities (5.1.2.1)**
Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more details, see *The DLIB runtime environment*, page 45.

**Diagnostic printed by the assert function (7.2.1.1)**
The `assert()` function prints:

```
filename:linernr expression -- assertion failed
```
when the parameter evaluates to zero.
Representation of the floating-point status flags (7.6.2.2)
For information about the floating-point status flags, see fenv.h, page 253.

Feraiseexcept raising floating-point exception (7.6.2.3)
For information about the feraiseexcept function raising floating-point exceptions, see fenv.h, page 253.

Strings passed to the setlocale function (7.11.1.1)
For information about strings passed to the setlocale function, see Locale, page 67.

Types defined for float_t and double_t (7.12)
The FLT_EVAL_METHOD macro can only have the value 0.

Domain errors (7.12.1)
No function generates other domain errors than what the standard requires.

Return values on domain errors (7.12.1)
Mathematics functions return a floating-point NaN (not a number) for domain errors.

Underflow errors (7.12.1)
Mathematics functions set errno to the macro ERANGE (a macro in errno.h) and return zero for underflow errors.

fmod return value (7.12.10.1)
The fmod function returns a floating-point NaN when the second argument is zero.

The magnitude of remqu0 (7.12.10.3)
The magnitude is congruent modulo INT_MAX.

signal() (7.14.1.1)
The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See Signal and raise, page 70.
**NULL macro (7.17)**

The NULL macro is defined to 0.

**Terminating newline character (7.19.2)**

STDOUT stream functions recognize either newline or end of file (EOF) as the terminating character for a line.

**Space characters before a newline character (7.19.2)**

Space characters written to a stream immediately before a newline character are preserved.

**Null characters appended to data written to binary streams (7.19.2)**

No null characters are appended to data written to binary streams.

**File position in append mode (7.19.3)**

The file position is initially placed at the beginning of the file when it is opened in append-mode.

**Truncation of files (7.19.3)**

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See File input and output, page 66.

**File buffering (7.19.3)**

An open file can be either block-buffered, line-buffered, or unbuffered.

**A zero-length file (7.19.3)**

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

**Legal file names (7.19.3)**

The legality of a filename depends on the application-specific implementation of the low-level file routines.

**Number of times a file can be opened (7.19.3)**

Whether a file can be opened more than once depends on the application-specific implementation of the low-level file routines.
Multibyte characters in a file (7.19.3)
The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

remove() (7.19.4.1)
The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See File input and output, page 66.

rename() (7.19.4.2)
The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See File input and output, page 66.

Removal of open temporary files (7.19.4.3)
Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

Mode changing (7.19.5.4)
freopen closes the named stream, then reopens it in the new mode. The streams stdin, stdout, and stderr can be reopened in any new mode.

Style for printing infinity or NaN (7.19.6.1, 7.24.2.1)
The style used for printing infinity or NaN for a floating-point constant is inf and nan (INF and NAN for the  conversion specifier), respectively. The n-char-sequence is not used for nan.

%p in printf() (7.19.6.1, 7.24.2.1)
The argument to a %p conversion specifier, print pointer, to printf() is treated as having the type void *. The value will be printed as a hexadecimal number, similar to using the %x conversion specifier.

Reading ranges in scanf (7.19.6.2, 7.24.2.1)
A – (dash) character is always treated as a range symbol.

%p in scanf (7.19.6.2, 7.24.2.2)
The %p conversion specifier, scan pointer, to scanf() reads a hexadecimal number and converts it into a value with the type void *.
File position errors (7.19.9.1, 7.19.9.3, 7.19.9.4)
On file position errors, the functions fgetpos, ftell, and fsetpos store EFPOS in errno.

An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)
An n-char-sequence after a NaN is read and ignored.

errno value at underflow (7.20.1.3, 7.24.4.1.1)
errno is set to ERANGE if an underflow is encountered.

Zero-sized heap objects (7.20.3)
A request for a zero-sized heap object will return a valid pointer and not a null pointer.

Behavior of abort and exit (7.20.4.1, 7.20.4.4)
A call to abort() or _Exit() will not flush stream buffers, not close open streams, and
not remove temporary files.

Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)
The termination status will be propagated to __exit() as a parameter. exit() and
_exit() use the input parameter, whereas abort uses EXIT_FAILURE.

The system function return value (7.20.4.6)
The system function is not supported.

The time zone (7.23.1)
The local time zone and daylight savings time must be defined by the application. For
more information, see Time, page 71.

Range and precision of time (7.23)
The implementation uses signed long for representing clock_t and time_t, based
at the start of the year 1970. This gives a range of approximately plus or minus 69 years
in seconds. However, the application must supply the actual implementation for the
functions time and clock. See Time, page 71.

clock() (7.23.2.1)
The application must supply an implementation of the clock function. See Time, page
71.
%Z replacement string (7.23.3.5, 7.24.5.1)
By default, ":" is used as a replacement for %Z. Your application should implement the time zone handling. See Time, page 71.

Math functions rounding mode (F.9)
The functions in math.h honor the rounding direction mode in FLT-ROUNDS.

J.3.13 Architecture

Values and expressions assigned to some macros (5.2.4.2, 7.18.2, 7.18.3)
There are always 8 bits in a byte.
MB_LEN_MAX is at the most 6 bytes depending on the library configuration that is used.
For information about sizes, ranges, etc for all basic types, see Data representation, page 189.
The limit macros for the exact-width, minimum-width, and fastest minimum-width integer types defined in stdint.h have the same ranges as char, short, int, long, and long long.
The floating-point constant FLT_ROUNDS has the value 1 (to nearest) and the floating-point constant FLT_EVAL_METHOD has the value 0 (treat as is).

The number, order, and encoding of bytes (6.2.6.1)
See Data representation, page 189.

The value of the result of the sizeof operator (6.5.3.4)
See Data representation, page 189.

J.4 Locale

Members of the source and execution character set (5.2.1)
By default, the compiler accepts all one-byte characters in the host’s default character set. If the compiler option --enable_multibytes is used, the host multibyte characters are accepted in comments and string literals as well.
The meaning of the additional character set (5.2.1.2)

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

Shift states for encoding multibyte characters (5.2.1.2)

Using the compiler option `--enable_multibytes` enables the use of the host’s default multibyte characters as extended source characters.

Direction of successive printing characters (5.2.2)

The application defines the characteristics of a display device.

The decimal point character (7.1.1)

The default decimal-point character is a ‘.’. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

Printing characters (7.4, 7.25.2)

The set of printing characters is determined by the chosen locale.

Control characters (7.4, 7.25.2)

The set of control characters is determined by the chosen locale.

Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

The sets of characters tested are determined by the chosen locale.

The native environment (7.1.1.1)

The native environment is the same as the "C" locale.

Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

The collation of the execution character set (7.21.4.3, 7.24.4.4.2)

The collation of the execution character set is determined by the chosen locale.
Message returned by strerror (7.21.6.2)

The messages returned by the `strerror` function depending on the argument is:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EZERO</td>
<td>no error</td>
</tr>
<tr>
<td>EDOM</td>
<td>domain error</td>
</tr>
<tr>
<td>ERANGE</td>
<td>range error</td>
</tr>
<tr>
<td>EFPOS</td>
<td>file positioning error</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>multi-byte encoding error</td>
</tr>
<tr>
<td>&lt;0</td>
<td></td>
</tr>
<tr>
<td>all others</td>
<td>error nnn</td>
</tr>
</tbody>
</table>

Table 49: Message returned by strerror()—IAR DLIB library
basic type names, using in preprocessor expressions
_________________________ 172
basic_template_matching (pragma directive) .................. 215
using ............................................ 121
batch files, error return codes ............................ 150
__bcd_add_long (intrinsic function) ................. 232
__bcd_add_long_long (intrinsic function) ............ 232
__bcd_add_short (intrinsic function) ........... 232
__bic_SR_register (intrinsic function) ....... 233
__bic_SR_register_on_exit (intrinsic function). 233
binary streams .................................... 279
bis_nmi_iel (pragma directive) ............... 215
__bis_SR_register (intrinsic function) ... 234
__bis_SR_register_on_exit (intrinsic function) 234
bit negation ....................................... 141

C

C and C++ linkage .................................. 93
C/C++ calling convention. See calling convention
C header files .................................... 249
C language, overview .............................. 105
call frame information ............................. 100
in assembler list file .............................. 89
in assembler list file (-Ia) ...................... 170
call stack ........................................ 100
callee-save registers, stored on stack ............ 20
calling convention
C++, requiring C linkage ....................... 90
in compiler .................................... 91
overriding default (__cc_version1) ............ 205
calloc (library function) ...................... 21
See also heap
can_instantiate (pragma directive) ............ 277
cassert (library header file) ................ 252
cast operators
in Extended EC++ ............................ 114, 123
missing from Embedded C++ ............... 114
casting
between pointer types .......................... 18
of pointers and integers ...................... 195
pointers to integers, language extension .... 110
ccomplex (library header file) ............. 252
cctype (DLIB header file) ..................... 252
__cc_version1 (extended keyword) ............ 205
__cc_version2 (extended keyword) .......... 206
cerrno (DLIB header file) ..................... 252
cfenv (library header file) ................... 252
CFI (assembler directive) .................... 100
CFI_COMMON (call frame information macro) 103
CFI_NAMES (call frame information macro) ... 103
c.m43 (CFI header example file) ............. 103
cfloat (DLIB header file) ..................... 252
char (data type) .................................. 190
changing default representation (--char_is_signed) 159
changing representation (--char_is_unsigned) 159
implementation-defined behavior ............. 272
signed and unsigned .......................... 191
character set, implementation-defined behavior .. 270
characters, implementation-defined behavior of .. 271
character-based I/O
in CLIB ........................................ 79
in DLIB ......................................... 63
overriding in runtime library ................ 46, 54
--char_is_signed (compiler option) ........ 159

IAR C/C++ Compiler
Reference Guide for MSP430
configuration
   basic project settings .................................... 5
   __low_level_init ...................................... 61
configuration symbols
   for file input and output .......................... 66
   for locale ........................................... 68
   for printf and scanf ................................ 65
   for strtod ......................................... 71
   in library configuration files ............. 57, 62
   consistency, module .................................. 73
   const
      declaring objects ................................ 200
      non-top level ..................................... 111
      constants, placing in named segment .... 216
   constseg (pragma directive) .................. 216
   const_cast (cast operator) ................. 114
   control characters,
   implementation-defined behavior .......... 283
   conventions, used in this guide .......... xxvi
   copy initialization, of segments .......... 43
   copyright notice .................................. 43
   __CORE__ (predefined symbol) ........... 242
   core
      identifying .................................... 242
      specifying on command line ............... 160
   --core (compiler option) ....................... 160
   __core (runtime model attribute) .......... 75
   __cplusplus (predefined symbol) ....... 242
   csetjmp (DLIB header file) ............ 252
   csignal (DLIB header file) ............ 252
   cspsy_support (pragma directive) .... 277
   CSTACK (segment) ................................. 260
   example ............................................... 38
   See also stack
   CSTART (segment) .................................. 41, 260
   cstartup (system startup code) .......... 41, 81
      customizing .................................... 62
      overriding in runtime library .......... 46, 54
   cstartup.s43 ....................................... 57
   cstdarg (DLIB header file) ............ 252
   cstdbool (DLIB header file) ............ 252
   cstddef (DLIB header file) ............ 252
   cstdio (DLIB header file) ............... 252
   cstdlib (DLIB header file) ............ 252
   cstring (DLIB header file) .............. 252
   ctgmath (library header file) ........ 252
   ctime (DLIB header file) ................. 252
   ctype.h (library header file) ......... 249, 254
   cwctype.h (library header file) ....... 252
   ?C_EXIT (assembler label) ............... 83
   ?C_GETCHAR (assembler label) .......... 83
   C_INCLUDE (environment variable) ....... 148
   ?C_PUTCHAR (assembler label) .......... 83
   C-SPY
      interface to system termination ........ 61
      low-level interface (CLIB) ............... 82
   STL container support ....................... 123
   C++
   See also Embedded C++ and Extended Embedded C++
      absolute location ................................ 133
      calling convention ................................ 90
      dynamic initialization in .......... 43
      features excluded from ECPP ...... 113
      header files .................................... 250
      language extensions ......................... 124
      special function types ..................... 30
      static member variables .................... 133
      support for .................................... 3
   C++ names, in assembler code .................. 91
   C++ objects, placing in memory type ........ 20
   C++ terminology .................................. xxvi
   C++-style comments ......................... 105
   --c89 (compiler option) ................. 159

D
   -D (compiler option) ......................... 160
   --data_model (compiler option) ............. 161
data
alignment of ............................................... 189
different ways of storing .................................. 13
data memory attributes, using ..................................... 16
data models .................................................. 14
data memory attributes, using ..................................... 16
configuration .................................................. 7
identifying (__DATA_MODEL__) .................................. 242
data pointers ................................................... 195
data segments ................................................... 35
data types ..................................................... 190
avoiding signed ................................................ 127
floating point .................................................. 193
in C++ .......................................................... 200
integers ........................................................ 111
data alignment (pragma directive) ......................... 217
data alignment (pragma directive) ......................... 217
__DATA_MODEL__ (predefined symbol) ....................... 242
__data16 (extended keyword) ................................ 206
data16 (memory type) ........................................ 16
DATA16_AC (segment) ........................................ 263
DATA16_AN (segment) ........................................ 261
DATA16_C (segment) ........................................... 264
DATA16_HEAP (segment) ...................................... 264
DATA16_ID (segment) .......................................... 264
DATA16_P (segment) .......................................... 265
DATA16_N (segment) .......................................... 265
DATA16_I (segment) .......................................... 265
DATA16_ZONE (segment) ...................................... 265
DATA16_P (segment) .......................................... 265
__data16_read_short (intrinsic function) ................... 235
__data16_read_long (intrinsic function) .................... 235
__data16_read_short (intrinsic function) ................... 235
__data16_size_t .............................................. 119
__data16_write_char (intrinsic function) ................... 236
__data16_write_long (intrinsic function) .................... 236
__data16_write_short (intrinsic function) ................... 236
DATA20_Z (segment) .......................................... 266
__DATE__ (predefined symbol) ................................ 242
data (library function), configuring support for .......... 71
data segments ................................................... 35
data block (call frame information) ......................... 100
debug information, including in object file .................. 162
declaration of ................................................ 189
declarations and statements, mixing ......................... 105
declarations and statements, mixing ......................... 105
declare_type_info (pragma directive) ....................... 277
delete (keyword) ............................................. 21
delete operator (extended EC++) ............................. 119
dependencies (compiler option) ............................... 162
deco (STL header file) ......................................... 251
diagnostic messages .......................................... 151
diagnostic messages .......................................... 151
destructors and interrupts, using ............................. 124
declassifying as compilation errors ........................... 163
declassifying as compilation remarks ......................... 163
data types ..................................................... 190
decimal point, implementation-defined behavior ............ 283
data memory attributes, using ..................................... 16
diffent ways of storing ....................................... 13
located, declaring extern ..................................... 133
placing ......................................................... 131, 184, 217, 257
storage ........................................................ 13
verifying linked result ........................................ 43
data block (call frame information) ......................... 100
data memory attributes, using ..................................... 16
data models ................................................... 14
configuration ................................................... 7
identifying (__DATA_MODEL__) ................................ 242
data pointers ................................................... 195
data segments ................................................... 35
data types ..................................................... 190
avoiding signed ................................................ 127
floating point .................................................. 193
in C++ .......................................................... 200
empty .......................................................... 111
in for loops .................................................... 105
Kernighan & Ritchie .......................................... 141
of functions ................................................... 92
declarations and statements, mixing ......................... 105
define_type_info (pragma directive) ....................... 277
delay_cycles (intrinsic function) ............................ 236
delete operator (extended EC++) ............................. 119
delete (keyword) ............................................. 21
dedependencies (compiler option) ............................. 162
deco (STL header file) ......................................... 251
destructors and interrupts, using ............................. 124
classifying as compilation errors ............................. 163
classifying as compilation remarks ........................... 163

data
alignment of ............................................... 189
different ways of storing .................................. 13
data memory attributes, using ..................................... 16
data models ................................................... 14
data memory attributes, using ..................................... 16
configuration ................................................... 7
identifying (__DATA_MODEL__) ................................ 242
data pointers ................................................... 195
data segments ................................................... 35
data types ..................................................... 190
avoiding signed ................................................ 127
floating point .................................................. 193
in C++ .......................................................... 200

Index

289
classifying as compiler warnings .................. 164
  disabling compiler warnings .................. 177
  disabling wrapping of in compiler ............. 177
  enabling compiler remarks .................... 183
  listing all used by compiler .................. 164
  suppressing in compiler ...................... 164
--diagnostics_tables (compiler option) ........ 164
diagnostics, implementation-defined behavior .... 269
diag_default (pragma directive) ................. 218
--diag_error (compiler option) .................. 163
diag_error (pragma directive) ................... 218
--diag_remark (compiler option) ................ 163
diag_remark (pragma directive) ................. 218
--diag_suppress (compiler option) ............... 164
diag_suppress (pragma directive) ............... 219
diag_warning (pragma directive) ................. 219
DIFUNCT (segment) ................................ 43, 266
DINT (assembler instruction) ..................... 236
directives
  function for static overlay ................... 98
    pragma .................................... 11, 213
directory, specifying as parameter .............. 154
__disable_interrupt (intrinsic function) ........ 236
--discard_unused_publics (compiler option) .... 165
do_not_instantiate (pragma directive) .......... 277
domain errors, implementation-defined behavior .. 278
_DOUBLE__ (predefined symbol) ................... 242
--double (compiler option) ..................... 166
double (data type) ................................ 193
  avoiding .................................. 127
  configuring size of floating-point type ........ 7
  identifying size of (__DOUBLE__) ............. 242
  __double__ (runtime model attribute) .......... 75
do_not instantiate (pragma directive) .......... 277
dynamic initialization .......................... 57, 81
  in C++ ................................... 43
dynamic memory ................................ 21
E
-e (compiler option) .......................... 167
early_initialization (pragma directive) ........ 277
--ec++ (compiler option) ...................... 167
EC++ header files ................................ 250
edition, of this guide ......................... ii
--eecc++ (compiler option) .................. 167
EINT (assembler instruction) .................... 236
Embedded C++ ................................ 113
  differences from C++ ......................... 113
  enabling .................................. 167
  function linkage ........................... 93
  language extensions ........................ 113
  overview .................................. 113
Embedded C++ Technical Committee ................ xxvi
embedded systems, IAR special support for ........ 10
__embedded_cplusplus (predefined symbol) ....... 243
__enable_interrupt (intrinsic function) ........ 236
--enable_multibytes (compiler option) .......... 168
entry label, program ........................ 58
enumerations, implementation-defined behavior ... 275
enums
  data representation ........................ 191
  forward declarations of .................... 110

document conventions, library .................. 247
environment .......................................................... 270
runtime (CLIB) ..................................................... 77
runtime (DLIB) ..................................................... 45
environment names, implementation-defined behavior ... 271
equivalent keywords .............................................. 201

extended command line file
__exit (library function) ............................................ 60
_exit (library function) ............................................. 60
exit (library function) .............................................. 60
implementation-defined behavior .................................. 281
__exit (library function) ............................................ 60
_exit (library function) ............................................. 60
export keyword, missing from Extended EC++ ......... 120
extended command line file
for compiler ......................................................... 168
passing options ...................................................... 147
Extended Embedded C++ ........................................... 114
enabling .............................................................. 167
standard template library (STL).............................. 251
extended keywords ................................................... 201
enabling (-e) ......................................................... 167
overview .............................................................. 10

summary .............................................................. 204
syntax ................................................................. 17
object attributes ................................................... 204
type attributes on data objects .................................... 202
type attributes on data pointers .................................... 203
type attributes on functions ...................................... 203
__ramfunc ............................................................ 30
extern "C" linkage .................................................. 118

f
-f (compiler option) ................................................. 168
fatal error messages .............................................. 152
defopen, in stdio.h ............................................... 253
feitrapdisable ........................................................ 253
feitrapenable ......................................................... 253
FENV_ACCESS, implementation-defined behavior .......... 274
fenv.h (library header file) ....................................... 249, 252
additional C functionality ........................................ 253
fgetpos (library function), implementation-defined behavior .................................................. 281
field width, library support for .................................... 80
__FILE__ (predefined symbol) ..................................... 243
file buffering, implementation-defined behavior .......... 279
file dependencies, tracking ....................................... 162
file paths, specifying for #include files ....................... 169
file position, implementation-defined behavior ............ 279
file (zero-length), implementation-defined behavior ...... 279
filename
extension for linker configuration file ......................... 33
search procedure for .............................................. 148
specifying as parameter ............................................ 154
filenames (legal), implementation-defined behavior .... 279
fileno, in stdio.h ................................................... 253
files, implementation-defined behavior
handling of temporary ............................................. 280
multibyte characters in ............................................ 280
opening ............................................................... 279
float (data type) ..................................................... 193
floating-point constants
  hexadecimal notation ............................................. 105
  hints ........................................................................ 128
floating-point environment, accessing or not ............... 229
floating-point expressions
  contracting or not ..................................................... 229
  using in preprocessor extensions ................................. 172
floating-point format .................................................. 193
  hints .......................................................................... 127–128
  implementation-defined behavior ................................. 273
  special cases ............................................................. 194
  32-bits ....................................................................... 194
  64-bits ....................................................................... 194
floating-point numbers, library support for .............. 80
floating-point status flags ........................................... 253
floating-point type, configuring size of double .......... 7
float.h (library header file) .......................................... 249, 254
FLT_EVAL_METHOD, implementation-defined behavior .......... 273, 278, 282
FLT_ROUNDS, implementation-defined behavior ............ 273, 282
for loops, declarations in ............................................. 105
formats
  floating-point values ................................................ 193
  standard IEEE (floating point) ..................................... 193
  __formatted_write (library function) ......................... 79
  FP_CONTRACT, implementation-defined behavior .......... 274
  fragmentation, of heap memory ................................ 22
  free (library function). See also heap ......................... 21
  fsetpos (library function), implementation-defined behavior ........................................ 281
  ifstream (library function), implementation-defined behavior ........................................ 281
  ftell (library function), implementation-defined behavior ........................................ 281
Full DLIB (library header file) ..................................... 250
__func__ (predefined symbol) ........................................ 112, 243
FUNCALL (assembler directive) .................................... 98
__FUNCTION__ (predefined symbol) .................. 112, 243
function calls
  calling convention ................................................... 91
  stack image after ..................................................... 96
function declarations, Kernighan & Ritchie ................. 141
function directives for static overlay ......................... 98
function execution, in RAM ........................................ 30
function inlining (compiler transformation) ................ 137
  disabling (--no_inline) ......................................... 174
  function pointers ................................................... 195
  function prototypes ................................................. 140
  enforcing ............................................................... 183
  function return addresses ....................................... 97
  function template parameter deduction (extended EC++) .... 121
  function type information, omitting in object output .... 178
  FUNCTION (assembler directive) ................................. 98
  function (pragma directive) ..................................... 277
  functional (STL header file) ..................................... 251
functions ................................................................. 23
  C++ and special function types ................................ 30
  declaring .............................................................. 92, 140
  inlining ................................................................. 105, 137, 140, 220
  interrupt ............................................................... 23, 26
  intrinsic .............................................................. 85, 140
  monitor ................................................................. 26
  omitting type info .................................................. 178
  parameters ........................................................... 94
  placing in memory ................................................ 131, 133, 184
  recursive
    avoiding ............................................................. 140
    storing data on stack ........................................... 21
    reentrancy (DLIB) ................................................ 248
    related extensions .............................................. 23
    return values from .............................................. 96
    special function types ........................................ 23
    verifying linked result ........................................ 43

G

getchar (library function) ........................................... 79
getenv (library function), configuring support for ........ 69
getw, in stdio.h ....................................................... 253
getzone (library function), configuring support for ....... 71
getzone.c ................................................. 71
__get_interrupt_state (intrinsic function) ................. 237
__get_R4_register (intrinsic function) .................... 238
__get_R5_register (intrinsic function) .................... 238
__get_SP_register (intrinsic function) ................. 238
__get_SR_register (intrinsic function) ................. 238
__get_SR_register_on_exit (intrinsic function) ....... 238
global variables, initializing ................................ 37
guidelines, reading ........................................ xxiii
Harbison, Samuel P .................................. xxvi
hardware multiplier ........................................ 72
enabled on command line .................................. 173
located ..................................................... 173
hardware support in compiler ................................ 45
hash_map (STL header file) ................................ 251
hash_set (STL header file) ................................ 251
hdrstop (pragma directive) .................................. 277
header files
  C ......................................................... 249
  C++ ..................................................... 250
  EC++ .................................................... 250
  library .................................................. 247
  special function registers ................................ 142
  STL ..................................................... 251
  DLib_Defaults.h ....................................... 249
  dl430libname.h ......................................... 57
  including stdbool.h for bool ........................... 190
  including stddef.h for wchar_t ........................ 191
header names, implementation-defined behavior .......... 276
__header_context (compiler option) ....................... 169
heap
  DLIB support for multiple ................................ 72
dynamic memory ............................................ 21
  segments for ........................................... 39
  storing data ............................................ 14
  heap segments
    CLIB ................................................... 40
    DATA16_HEAP (segment) .................................. 261
    DATA20_HEAP (segment) .................................. 264
    DLIB .................................................. 39
    placing ................................................ 40
heap size
  and standard I/O ....................................... 41
  changing default ....................................... 40
heap (zero-sized), implementation-defined behavior ..... 281
  hints
    for good code generation ................................ 139
    implementation-defined behavior ....................... 274
    using efficient data types ............................ 127
IDE
  -I (compiler option) .................................... 169
  IAR Command Line Build Utility ........................ 56
  IAR Systems Technical Support ........................ 152
  iarbuild.exe (utility) .................................. 56
  __IAR_SYSTEMS_ICC__ (predefined symbol) ........... 243
  iccbutl.h (library header file) ......................... 254
  __ICC430__ (predefined symbol) ........................ 243
icons, in this guide ........................................ xxvii
IDE
  building a library from ................................ 56
  building applications from, an overview ............. 4
  identifiers, implementation-defined behavior .......... 271
  IEEE format, floating-point values ................... 193
  implementation-defined behavior ....................... 269
  important_typedef (pragma directive) ............... 277
  include files
    including before source files ....................... 180
    specifying .......................................... 148
    include_alias (pragma directive) .................. 220
    infinity ............................................. 194
    infinity (style for printing), implementation-defined
      behavior ........................................... 280
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFO (segment)</td>
<td>266</td>
</tr>
<tr>
<td>INFOA (segment)</td>
<td>267</td>
</tr>
<tr>
<td>INFOB (segment)</td>
<td>267</td>
</tr>
<tr>
<td>INFOC (segment)</td>
<td>267</td>
</tr>
<tr>
<td>INFOD (segment)</td>
<td>267</td>
</tr>
<tr>
<td>information memory, linker segment for</td>
<td>267</td>
</tr>
<tr>
<td>inheritance, in Embedded C++</td>
<td>113</td>
</tr>
<tr>
<td>initialization</td>
<td></td>
</tr>
<tr>
<td>dynamic</td>
<td>57, 81</td>
</tr>
<tr>
<td>single-value</td>
<td>111</td>
</tr>
<tr>
<td>initialized data segments</td>
<td>37</td>
</tr>
<tr>
<td>initializers, static</td>
<td>110</td>
</tr>
<tr>
<td>inline assembler</td>
<td>87, 106</td>
</tr>
<tr>
<td>avoiding</td>
<td>140</td>
</tr>
<tr>
<td>See also assembler language interface</td>
<td></td>
</tr>
<tr>
<td>inline functions</td>
<td>105</td>
</tr>
<tr>
<td>in compiler</td>
<td>137</td>
</tr>
<tr>
<td>inline (pragma directive)</td>
<td>220</td>
</tr>
<tr>
<td>inlining functions, implementation-defined behavior</td>
<td>275</td>
</tr>
<tr>
<td>instantiate (pragma directive)</td>
<td>277</td>
</tr>
<tr>
<td>integers</td>
<td>190</td>
</tr>
<tr>
<td>casting</td>
<td>195</td>
</tr>
<tr>
<td>implementation-defined behavior</td>
<td>273</td>
</tr>
<tr>
<td>intptr_t</td>
<td>196</td>
</tr>
<tr>
<td>ptdiff_t</td>
<td>196</td>
</tr>
<tr>
<td>size_t</td>
<td>196</td>
</tr>
<tr>
<td>intptr_t</td>
<td>196</td>
</tr>
<tr>
<td>integral promotion</td>
<td>141</td>
</tr>
<tr>
<td>internal error</td>
<td>152</td>
</tr>
<tr>
<td>__interrupt (extended keyword)</td>
<td>24, 207</td>
</tr>
<tr>
<td>using in pragma directives</td>
<td>230</td>
</tr>
<tr>
<td>interrupt functions</td>
<td>23</td>
</tr>
<tr>
<td>placement in memory</td>
<td>42</td>
</tr>
<tr>
<td>interrupt state, restoring</td>
<td>239</td>
</tr>
<tr>
<td>interrupt vector</td>
<td>24</td>
</tr>
<tr>
<td>specifying with pragma directive</td>
<td>230</td>
</tr>
<tr>
<td>interrupt vector table</td>
<td>24</td>
</tr>
<tr>
<td>in linker configuration file</td>
<td>42</td>
</tr>
<tr>
<td>INTVEC segment</td>
<td>268</td>
</tr>
<tr>
<td>interrupts</td>
<td></td>
</tr>
<tr>
<td>disabling</td>
<td>207</td>
</tr>
<tr>
<td>during function execution</td>
<td>26</td>
</tr>
<tr>
<td>processor state</td>
<td>20</td>
</tr>
<tr>
<td>using with EC++ destructors</td>
<td>124</td>
</tr>
<tr>
<td>intptr_t (integer type)</td>
<td>196</td>
</tr>
<tr>
<td>__ intrinsic (extended keyword)</td>
<td>207</td>
</tr>
<tr>
<td>intrinsic functions</td>
<td>140</td>
</tr>
<tr>
<td>overview</td>
<td>85</td>
</tr>
<tr>
<td>summary</td>
<td>231</td>
</tr>
<tr>
<td>intrinsics.h (header file)</td>
<td>231</td>
</tr>
<tr>
<td>inttypes.h (library header file)</td>
<td>249</td>
</tr>
<tr>
<td>INTVEC (segment)</td>
<td>42, 268</td>
</tr>
<tr>
<td>intwri.c (library source code)</td>
<td>80</td>
</tr>
<tr>
<td>invocation syntax</td>
<td>147</td>
</tr>
<tr>
<td>iomanip (library header file)</td>
<td>250</td>
</tr>
<tr>
<td>ios (library header file)</td>
<td>250</td>
</tr>
<tr>
<td>iosfwd (library header file)</td>
<td>250</td>
</tr>
<tr>
<td>istream (library header file)</td>
<td>250</td>
</tr>
<tr>
<td>ISO/ANSI C++ features excluded from EC++</td>
<td>113</td>
</tr>
<tr>
<td>iso646.h (library header file)</td>
<td>249</td>
</tr>
<tr>
<td>ISR_CODE (segment)</td>
<td>268</td>
</tr>
<tr>
<td>using</td>
<td>42</td>
</tr>
<tr>
<td>italic style, in this guide</td>
<td>xxvii</td>
</tr>
<tr>
<td>iterator (STL header file)</td>
<td>251</td>
</tr>
<tr>
<td>I/O module, overriding in runtime library</td>
<td>46, 54</td>
</tr>
<tr>
<td>I/O, character-based</td>
<td>79</td>
</tr>
</tbody>
</table>

K

keep_definition (pragma directive) | 277 |
Kernighan & Ritchie function declarations | 141 |
| disallowing | 183 |
Kernighan, Brian W. | xxvi |
| keywords | 201 |
| extended, overview of | 10 |
long double (data type) ........................................ 193
long float (data type), synonym for double ........ 110
long long (data type)
  avoiding ........................................ 127
  restrictions .................................... 190
loop overhead, reducing .............................. 177
loop unrolling (compiler transformation) ........ 137
  disabling .................................... 177
loop-invariant expressions ......................... 138
__low_level_init ..................................... 58
  customizing ................................... 61
low_level_init.c ...................................... 57, 81
__low_power_mode_n (intrinsic function) .......... 239
__low_power_mode_off_on_exit (intrinsic function) . 239
low-level processor operations .................. 107, 231
  accessing .................................... 85
__seek (library function) ......................... 81

M

macros
  embedded in #pragma optimize .................... 224
  ERANGE (in errno.h) ................................ 278
  inclusion of assert ................................ 244
  NULL, implementation-defined behavior .......... 279
  substituted in #pragma directives ............. 107
  variadic ...................................... 105
main (function), implementation-defined behavior .. 270
malloc (library function)
  See also heap .................................. 21
Mann, Bernhard ........................................ xxi
map (STL header file) .............................. 251
  map, linker .................................. 44
math functions rounding mode, ..................... 282
implementation-defined behavior ................ 282
math.h (library header file) ..................... 250, 254
MB_LEN_MAX, implementation-defined behavior ...... 282
Medium data model ................................ 15
__medium_write (library function) ................ 80
memory
  accessing .................................... 7, 16
  allocating in C++ ............................ 21
  dynamic ..................................... 21
  heap ........................................ 21
  non-initialized .............................. 144
  RAM, saving ................................ 140
  releasing in C++ ............................ 21
  stack ........................................ 20
  saving ...................................... 140
  used by global or static variables ............. 14
  memory consumption, reducing ................. 79
  memory layout, MSP430 and MSP430X ........... 13
  memory management, type-safe .................. 113
  memory map
    customizing the linker configuration file for .. 33
  memory placement
    using pragma directive ..................... 17
    using type definitions ..................... 17, 203
  memory segment. See segment
  memory types .................................. 15
  C++ .......................................... 20
  placing variables in .......................... 20
  pointers .................................... 18
  specifying .................................. 16
  structures .................................. 18
  summary ..................................... 17
  memory (pragma directive) ..................... 277
  memory (STL header file) ..................... 251
  __memory_of, operator ......................... 117
  message (pragma directive) ................... 223
  messages
    disabling .................................. 185
    forcing .................................... 223
    -mfc (compiler option) ..................... 171
    -migration_preprocessor_extensions (compiler option) . 172
    -misrac (compiler option) ................ 157
    -misrac_verbose (compiler option) ....... 157
    -misrac1998 (compiler option) ............ 157
--misrac2004 (compiler option) ....................... 157
mode changing, implementation-defined behavior .... 280
module consistency .................................... 73
  rmodel ........................................... 227
module map, in linker map file ........................ 44
module name, specifying ................................ 172
module summary, in linker map file .................... 44
--module_name (compiler option) .................... 172
__module_name (pragma directive) .................... 277
__monitor (extended keyword) ......................... 207
monitor functions .................................... 26, 207
MSP430
  information memory .................................. 267
MSP430 and MSP430X
  architecture ...................................... 6
  memory access ..................................... 7
  memory layout ...................................... 13
  supported devices .................................. 4
multibyte character support .......................... 168
multibyte characters, implementation-defined behavior ........................................... 271, 283
multiple inheritance
  missing from Embedded C++ .......................... 113
  missing from STL ................................... 114
--multiplier (compiler option) ...................... 173
--multiplier_location (compiler option) ............ 173
multi-file compilation ................................ 135
mutable attribute, in Extended EC++ ................. 114, 123
native environment, implementation-defined behavior ................... 283
NDEBUG (preprocessor symbol) .......................... 244
new operator (extended EC++) .......................... 119
new (keyword) ...................................... 21
new (library header file) ............................. 250
non-initialized variables, hints for .................... 144
non-scalar parameters, avoiding ....................... 140
NOP (assembler instruction) ........................... 239
__noret (extended keyword) ........................... 208
Normal DLIB (library configuration) .................... 62
Not a number (NaN) .................................. 195
--no_code_motion (compiler option) ................. 174
--no_cse (compiler option) .......................... 174
no_epilogue (pragma directive) ....................... 223
__no_init (extended keyword) ........................ 144, 208
--no_inline (compiler option) ....................... 174
__no_operation (intrinsic function) .................. 239
--no_path_in_file_macros (compiler option) ........ 175
no_pch (pragma directive) ........................... 277
--no_static_destruction (compiler option) .......... 175
--no_system_include (compiler option) .............. 175
--no_typedefs_in_diagnostics (compiler option) .... 176
--no_unroll (compiler option) ...................... 177
--no_warnings (compiler option) .................... 177
--no_wrap_diagnostics (compiler option) ............ 177
NULL .............................................. 255
  implementation-defined behavior .................... 279
numeric (STL header file) ........................... 251

N
  names block (call frame information) ............... 100
  namespace support
    in Extended EC++ ................................ 114, 123
    missing from Embedded C++ ......................... 114
  naming conventions ................................ xxviii
NaN
  implementation of .................................. 195
  implementation-defined behavior .................... 280

O
  -O (compiler option) ................................ 178
  -o (compiler option) ................................ 179
  object attributes ................................ 204
  object module name, specifying ..................... 172
  object_attribute (pragma directive) ............... 144, 223
  offsetof ........................................ 255
  --omit_types (compiler option) .................... 178
  once (pragma directive) ........................... 277
pack (pragma directive) ........................................... 197, 225
packed structure types ............................................. 197
parameters
  function ......................................................... 94
  hidden ........................................................... 94

non-scalar, avoiding .............................................. 140

operators
  See also @ (operator)
  __memory_of ........................................................ 117

optimization
  code motion, disabling ........................................... 174
  common sub-expression elimination, disabling ............... 174
  configuration ..................................................... 8
  disabling ............................................................ 137
  function inlining, disabling (--no_inline) ................. 174
  hints ................................................................. 139
  loop unrolling, disabling ...................................... 177
  specifying (-O) ................................................... 178
  techniques .......................................................... 137
  type-based alias analysis, disabling (--tbaa) ............... 176
  using inline assembler code .................................... 87
  using pragma directive ........................................ 224
  optimization levels ............................................. 136
  optimize (pragma directive) .................................... 224
  option parameters .............................................. 153
  options, compiler. See compiler options
    __op_code (intrinsic function) ................................. 239

Oram, Andy ........................................................... xxvi

output
  from preprocessor .............................................. 180
  specifying for linker ........................................... 5
  supporting non-standard ...................................... 80
  --output (compiler option) .................................... 179
  overhead, reducing .............................................. 137

polymorphism, in Embedded C++ ................................ 113

pointers
  casting ........................................................... 18, 195
  data ............................................................... 195
  function .......................................................... 195
  implementation-defined behavior .............................. 274

porting, code containing pragma directives ............................ 214

position independent code ........................................ 10
  __POSITION_INDEPENDENT_CODE__ (predefined symbol) ........ 243

pow (library function) ........................................... 71
powXp (library function) ........................................ 71

pragma directives ................................................ 11
  summary .......................................................... 213
  basic_template_matching, using ................................ 121
  for absolute located data ..................................... 132
  list of all recognized ......................................... 276
  pack ............................................................... 197
  type_attribute, using ......................................... 17
  Pragma (preprocessor operator) ................................ 105

precision arguments, library support for ............................ 80

See also @ (operator)

See compiler options................................................... 225

See also precision arguments, library support for ................. 80

See also type_attribute, using ................................... 17

See also type-based alias analysis, disabling (--tbaa) ............ 176

See also using inline assembler code ................................ 87

See also using pragma directive ................................... 224

See also from preprocessor ........................................ 180

See also specifying for linker .................................... 5

See also supporting non-standard ................................ 80

See also --output (compiler option) ................................ 179

See also overhead, reducing ...................................... 137

See also optimization ............................................. 136

See also optimize (pragma directive) ................................ 224

See also option parameters ...................................... 153

See also options, compiler. See compiler options
  __op_code (intrinsic function) .................................. 239

See also output
  from preprocessor .............................................. 180
  specifying for linker ........................................... 5
  supporting non-standard ...................................... 80
  --output (compiler option) .................................... 179

See also type-based alias analysis, disabling (--tbaa) ............ 176

See also using inline assembler code ................................ 87

See also using pragma directive ................................... 224

See also code motion, disabling ................................... 174

See also common sub-expression elimination, disabling ............ 174

See also configuration ............................................. 8

See also disabling ................................................... 137

See also function inlining, disabling (--no_inline) ............... 174

See also hints ........................................................ 139

See also loop unrolling, disabling ................................ 177

See also specifying (-O) ........................................... 178

See also techniques .................................................. 137

See also type-based alias analysis, disabling (--tbaa) ............ 176

See also using inline assembler code ................................ 87

See also using pragma directive ................................... 224

See also optimization levels ...................................... 136

See also optimize (pragma directive) ................................ 224

See also option parameters ...................................... 153

See also options, compiler. See compiler options
  __op_code (intrinsic function) .................................. 239

See also Oram, Andy .................................................. xxvi

See also ostream (library header file) ................................ 251

See also output
  from preprocessor .............................................. 180
  specifying for linker ........................................... 5
  supporting non-standard ...................................... 80
  --output (compiler option) .................................... 179

See also overhead, reducing ...................................... 137
predefined symbols

overview .................................. 11
summary .................................. 242
--predef_macro (compiler option) ....... 179
--preinclude (compiler option) ........ 180
--preprocess (compiler option) ........ 180
preprocessor
output .................................... 180
overview of ................................ 241
preprocessor directives
comments at the end of ...................... 111
implementation-defined behavior .......... 276
#pragma .................................... 213
preprocessor extensions
compatibility ................................ 172
__VA_ARGS__ ................................ 105
#pragma .................................... 245
preprocessor operator
__Pragma() ................................... 105
preprocessor symbols ....................... 242
defining .................................... 160
preserved registers ......................... 94
__PRETTY_FUNCTION__ (predefined symbol) ........ 243
primitives, for special functions .......... 23
print formatter, selecting .................. 51
printf (library function) ................. 50, 79
choosing formatter ......................... 50
configuration symbols ...................... 65
customizing ................................ 80
implementation-defined behavior .......... 280
selecting ................................... 80
__printf_args (pragma directive) ........ 226
printing characters, implementation-defined behavior ... 283
processor configuration .................... 6
processor operations
accessing ................................... 85
low-level ................................... 107, 231
program entry label ......................... 58
program termination, implementation-defined behavior .. 270
programming hints .......................... 139
__program_start (label) .................... 58
projects
basic settings for .......................... 5
setting up for a library .................... 56
prototypes, enforcing ....................... 183
ptrdiff_t (integer type) .................... 196, 255
PUBLIC (assembler directive) .............. 181
publication date, of this guide ............ ii
--public_equ (compiler option) .......... 181
public_equ (pragma directive) ............ 277
putchar (library function) ................. 79
putenv (library function), absent from DLIB ........ 69
putw, in stdio.h ................................ 253

Q

-Q (linker option) ......................... 43
QCC430 (environment variable) ............ 148
qualifiers
const and volatile ......................... 199
implementation-defined behavior .......... 275
queue (STL header file) ................... 251

R

-r (compiler option) ....................... 162
raise (library function), configuring support for ........ 70
raise.c .................................. 70
RAM
non-zero initialized variables ............ 37
saving memory ................................ 140
RAM execution ............................. 30
__ramfunc (extended keyword) ............. 209
range errors, in linker ..................... 43
__raw (extended keyword) .................. 209
example ................................... 25
__read (library function) ................ 67
customizing ............................... 63
read formatter, selecting .............................. 52, 81
reading guidelines ........................................ xxiii
reading, recommended ...................................... xxvi
read-only memory, placing data in .................... 210
realloc (library function)
  See also heap ............................................. 21
recursive functions
  avoiding .................................................. 140
  storing data on stack .................................... 21
--reduce_stack_usage (compiler option) ................ 181
reentrancy (DLIB) ........................................... 248
reference information, typographic convention ........ xxvii
register keyword, implementation-defined behavior .... 274
register parameters ....................................... 94–95
registered trademarks ................................... .ii
registers
  assigning to parameters .................................. 95
  callee-save, stored on stack ........................... 20
  for function returns ..................................... 96
  in assembler-level routines ............................. 91
  preserved ................................................... 94
R4
  excluding from use (--lock_R4) ......................... 171
  getting the value of (__get_R4_register) .............. 238
  reserving for register variables (--regvar_R4) ....... 181
  writing to (__set_R4_register) ......................... 240
R5
  excluding from use (--lock_R5) ......................... 171
  getting the value of (__get_R5_register) .............. 238
  reserving for register variables (--regvar_R5) ....... 182
  writing to (__set_R5_register) ......................... 240
scratch ...................................................... 93
SP
  getting the value of (__get_SP_register) .............. 238
  writing to (__set_SP_register) ......................... 240
SR
  getting the value of on exit ........................... 238
  getting the value of (__get_SR_register) .............. 238
_REGISTER_MODEL__ (predefined symbol) ............. 244
REGVAR_AN (segment) ....................................... 268
--regvar_r4 (compiler option) .......................... 181
--regvar_r5 (compiler option) .......................... 182
__reg_4 (runtime model attribute) ..................... 75
__reg_5 (runtime model attribute) ..................... 75
reinterpret_cast (cast operator) ....................... 114
--relaxed_fp (compiler option) ......................... 182
remark (diagnostic message) ............................ 151
  classifying for compiler ............................... 163
  enabling in compiler ................................... 183
--remarks (compiler option) ............................ 183
remove (library function) ................................ 67
  implementation-defined behavior ...................... 280
remquo, magnitude of ................................... 278
rename (library function) ................................ 67
  implementation-defined behavior ...................... 280
__ReportAssert (library function) ...................... 72
required (pragma directive) ............................. 226
--require_prototypes (compiler option) ................. 183
RESET (segment) .......................................... 268
return addresses .......................................... 97
return values, from functions .......................... 96
Ritchie, Dennis M. ......................................... xxvi
setting options .......................................... 10
runtime environment
  CLIB ...................................................... 77
  DLIB ...................................................... 45
  runtime libraries
    choosing .................................................. 9
    introduction ........................................... 247
    CLIB ...................................................... 77
    naming convention .................................... 78
Index

DLIB ................................................. 47
  choosing ........................................ 47
  customizing without rebuilding .......... 49
  naming convention ............................ 49
  overriding modules in ...................... 46, 54
runtime model attributes ...................... 73
runtime model definitions ..................... 227
runtime type information, missing from Embedded C++ . 114

R4. See registers
R5. See registers

S

---save_reg20 (compiler option) ................... 183
scanf (library function) .......................... 81
  choosing formatter ............................ 51
  configuration symbols ........................ 65
  implementation-defined behavior .......... 280
  _scanf_args (pragma directive) .............. 227
scratch registers .................................. 93
segment group name ................................ 36
segment map, in linker map file .............. 44
segment memory types, in XLINK ............... 32
segment names
  declaring ....................................... 228
  specifying .................................... 184
segment (pragma directive) ...................... 228
segments .......................................... 257
code ............................................. 41
data ............................................. 35
definition of ................................... 31
initialized data ................................... 37
introduction ....................................... 31
located data ...................................... 41
naming ............................................ 36
packing in memory ................................ 34
placing in sequence .............................. 34
static memory .................................... 35
summary ........................................... 257
too long for address range ....................... 43
too long, in linker ............................... 43
CODE ............................................. 42
  ISR_CODE, for interrupt functions (MSP430X) .... 42
  __segment_begin (extended operator) ........... 109
  __segment_end (extended operator) ............. 109
  __segment_size (extended operator) .......... 109
semaphores
  C example ....................................... 27
  C++ example .................................... 28
  operations on ................................... 207
set (STL header file) ............................. 250, 254
setlocale (library function) ................... 69
settings, basic for project configuration ...... 5
  __set_interrupt_state (intrinsic function) ... 239
  __set_R4_register (intrinsic function) ......... 240
  __set_R5_register (intrinsic function) ........ 240
  __set_SP_register (intrinsic function) ...... 240
severity level, of diagnostic messages ....... 151
  specifying .................................... 152
SFR
  accessing special function registers ........ 142
  declaring extern special function registers .... 133
shared object ..................................... 150
short (data type) ................................ 190
signal (library function)
  configuring support for ........................ 70
  implementation-defined behavior ............ 278
signals, implementation-defined behavior .... 270
  at system startup ................................ 271
signal.c .......................................... 70
signal.h (library header file) ................. 250
signed char (data type) ......................... 190–191
  specifying .................................... 159
signed int (data type) ............................ 190
signed long long (data type) .................... 190
signed long (data type) .......................... 190
signed short (data type) ........................ 190

C430-8
signed values, avoiding ........................................ 127
--silent (compiler option) ...................................... 185
silent operation, specifying in compiler ..................... 185
64-bits (floating-point format) ............................... 194
sizeof, using in preprocessor extensions ................... 172
size_t (integer type) ........................................... 196, 255
skeleton code, creating for assembler language interface .... 88
skeleton.s43 (assembler source output) ........................ 89
slist (STL header file) ......................................... 251
Small data model ................................................. 15
_small_write (library function) ............................... 80
source files, list all referred .................................. 169
space characters, implementation-defined behavior .......... 279
special function registers (SFR) .............................. 142
special function types .......................................... 23
overview .......................................................... 11
sprintf (library function) ....................................... 50, 79
choosing formatter .............................................. 50
customizing ....................................................... 80
SP. See registers
sstream (library header file) ................................ 251
choosing formatter .............................................. 51
sstream (library header file) ................................ 251
_stack (library header file) .................................... 251
stack advantages and problems using ........................ 21
changing default size of ....................................... 38
cleaning after function return ................................ 97
contents of ........................................................ 20
layout ............................................................... 95
saving space ....................................................... 140
size ................................................................. 39
stack parameters .................................................. 94–95
stack pointer ....................................................... 20
stack segment
CSTACK ............................................................ 263
placing in memory ............................................... 39
stack (STL header file) ......................................... 251
Standard C
library compliance with ....................................... 8, 247

specifying strict usage ............................................ 185
standard error, redirecting in compiler ....................... 179
standard input .................................................... 63
standard output .................................................. 63
specifying in compiler ......................................... 179
standard template library (STL)
in Extended EC++ ................................................ 114, 122, 251
missing from Embedded C++ .................................... 114
startup code
placement of ....................................................... 41
See also CSTART
startup system. See system startup
static data, in linker configuration file ....................... 38
static memory segments ......................................... 35
static overlay ..................................................... 98
static variables ................................................... 14
initializing .......................................................... 37
taking the address of ............................................ 139
static_cast (cast operator) ..................................... 114
status flags for floating-point ................................ 253
std namespace, missing from EC++ and Extended EC++. . 124
stdarg.h (library header file) ................................ 250, 254
stdbool.h (library header file) ................................ 190, 250, 254
__STDC__ (predefined symbol) ................................ 244
STDC CX_LIMITED_RANGE (pragma directive) ............. 228
STDC FENV_ACCESS (pragma directive) ....................... 229
STDC FP_CONTRACT (pragma directive) ...................... 229
__STDC_VERSION__ (predefined symbol) ..................... 244
stddef.h (library header file) ................................ 191, 250, 255
stderr ............................................................. 67, 179
stdexcept (library header file) ................................ 251
stdin .............................................................. 67
stdout .............................................................. 67
stdint.h (library header file) ................................ 250, 252
stdio.h (library header file) .................................. 250, 255
stdio.h, additional C functionality ........................ 253
stdlib.h (library header file) ................................ 250, 255
stdio.h ............................................................ 67, 179
implementation-defined behavior ............................ 279
Steele, Guy L......................................................... xxvi
STL .................................................. 122
strcasemkp, in string.h .......................... 254
strdup, in string.h .............................. 254
streambuf (library header file) .............. 251
streams
implementation-defined behavior ............. 270
.. supported in Embedded C++ .............. 114
strerror (library function), implementation-defined
.. behavior ........................................ 284
--strict (compiler option) ....................... 185
string (library header file) ................. 251
strings, supported in Embedded C++ ......... 114
string.h (library header file) ............... 250
strnlen, in string.h ........................... 254
strncasecmp, in string.h ....................... 254
String(h, additional C functionality .......... 254
stringbuf, in string.h ......................... 254
Stroustrup, Bjarne ................................ xxvi
strings, supported in Embedded C++ .......... 114
string (library header file) ................... 251
sttrod (library function), configuring support for .. 71
structure types
alignment ........................................ 197
layout of ........................................ 197
packed ............................................ 197
structures
aligning ........................................... 225
anonymous ....................................... 108, 129
implementation-defined behavior .......... 275
packing and unpacking .......................... 129
placing in memory type ....................... 18
subnormal numbers ............................... 193
__SUBVERSION__ (predefined symbol) ......... 244
support, technical ................................ 152
__swap_bytes (intrinsic function) ............ 240
switch statements, hints for using ............ 144
SWPB (assembler instruction) ................. 240
symbol names, using in preprocessor extensions 172
symbols
anonymous, creating ............................ 105
including in output ............................. 226
listing in linker map file ..................... 44
overview of predefined ....................... 11
preprocessor, defining ....................... 160
syntax
command line options ......................... 153
extended keywords ......................... 17, 202–204
invoking compiler ............................. 147
system function, implementation-defined behavior .. 271, 281
system startup
CLIB ............................................ 81
customizing .................................... 61
DLIB ............................................ 58
system termination
CLIB ............................................ 82
C-SPY interface to ............................. 61
DLIB ............................................ 60
system (library function), configuring support for .... 69
__SystemLibrary (runtime model attribute) .... 75
system_includes (pragma directive) .......... 277
--system_includes (compiler option) ........ 186
__task (extended keyword) ..................... 211
technical support, IAR Systems ............... 152
template support
.. in Extended EC++ ......................... 114, 120
.. missing from Embedded C++ ............ 113
Terminal I/O window
.. making available (CLIB) .................... 83
.. making available (DLIB) .................... 54
.. not supported when ....................... 55
terminal I/O, debugger runtime interface for .......... 53
terminal output, speeding up .................. 54
termination of system .......................... See system termination
termination status, implementation-defined behavior .. 281
terminology .................................. xxvi
tgmath.h (library header file) ............... 250
32-bits (floating-point format) .............. 194
this (pointer) .................................................. 90
class memory .................................................. 116
referred to a class object .................................. 116
__TIME__ (predefined symbol) ......................... 244
time zone (library function), implementation-defined
behavior ....................................................... 281
time (library function), configuring support for .......... 71
time-critical routines ......................................... 85, 107, 231
time.c ............................................................ 71
time.h (library header file) ................................ 250
tips, programming ............................................ 139
tools icon, in this guide ...................................... xxvii
trademarks ....................................................... ii
transformations, compiler ................................... 135
translation, implementation-defined behavior ............. 269
trap vectors, specifying with pragma directive ......... 230
type attributes ................................................ 201
specifying ....................................................... 230
type definitions, used for specifying memory storage .17, 203
type information, omitting .................................. 178
type qualifiers
const and volatile ............................................ 199
implementation-defined behavior ....................... 275
typedef
excluding from diagnostics ............................. 176
repeated ....................................................... 110
using in preprocessor extensions ....................... 172
type_attribute (pragma directive) ...................... 17, 230
type-based alias analysis (compiler transformation) .... 138
disabling ....................................................... 176
type-safe memory management ............................ 113
typographic conventions ................................... xxvii

U

UBROF
format of linkable object files ......................... 149
specifying, example of .................................. 5
uchar.h (library header file) ............................ 250
UCNs (universal character names), no support for .... 106
uintptr_t (integer type) ................................. 196
underflow errors, implementation-defined behavior .... 278
unions
anonymous .................................................... 108, 129
implementation-defined behavior .................... 275
universal character names, implementation-defined
behavior ....................................................... 276
unsigned char (data type) ................................. 190–191
changing to signed char .................................. 159
unsigned int (data type) ................................. 190
unsigned long (data type) ............................... 190
unsigned long (data type) ............................... 190
unsigned short (data type) ............................. 190
utility (STL header file) .................................. 251
variable type information, omitting in object output .... 178
variables
auto .......................................................... 20
defined inside a function .................................. 20
global, placement in memory ............................ 14
hints for choosing ......................................... 139
local. See auto variables
non-initialized .............................................. 144
omitting type info .......................................... 178
placing at absolute addresses .......................... 133
placing in named segments .............................. 133
static
placement in memory ..................................... 14
taking the address of .................................... 139
static and global, initializing ......................... 37
vector (pragma directive) ................................. 24, 230
vector (STL header file) .................................. 251
__VER__ (predefined symbol) ......................... 244
version
compiler ..................................................... 244
IAR Embedded Workbench ................................ ii
version1 calling convention ........................ 92
version2 calling convention ........................ 92
--vla (compiler option) ............................ 186
void, pointers to ................................... 110
volatile (keyword) .................................. 142
volatile, declaring objects ......................... 199–200

W

#warning message (preprocessor extension) ........ 245
warnings .............................................. 151
classifying in compiler ................................ 164
disabling in compiler ................................ 177
exit code in compiler ................................ 186
warnings icon, in this guide ........................ xxvii
warnings pragma directive .......................... 277
--warnings_affect_exit_code (compiler option) ... 150, 186
--warnings_are_errors (compiler option) .......... 187
wchar_t (data type), adding support for in C .. 191
wchar.h (library header file) ......................... 250, 252
wctype.h (library header file) ....................... 250
web sites, recommended ................................ xxvi
white-space characters, implementation-defined behavior 269
_customizing (library function) .................... 67
_write (library function) .............................. 80–81

X

XLINK errors

range error ........................................... 43
data too long ......................................... 43
XLINK segment memory types .......................... 32
xreportassert.c ........................................ 72

Symbols

_exit (library function) ............................... 60
_exit (library function) ............................... 60
formatted_write (library function) ............... 79
_medium_write (library function) .................. 80
_small_write (library function) .................... 80
.ALIGNOF_ (operator) ................................ 108
asm (language extension) ............................ 106
BASE_FILE_ (predefined symbol) ................. 242
bcd_add_long (intrinsic function) ............... 232
bcd_add_long_long (intrinsic function) ....... 232
bcd_add_short (intrinsic function) .............. 232
bic_SR_register (intrinsic function) .......... 233
bic_SR_register_on_exit (intrinsic function) . 233
bis_SR_register (intrinsic function) .......... 234
bis_SR_register_on_exit (intrinsic function) . 234
BUILD_NUMBER_ (predefined symbol) ............ 242
_cc_version1 (extended keyword) ................. 205
choosing calling convention ......................... 92
_cc_version2 (extended keyword) ................. 206
choosing calling convention ......................... 92
close (library function) ............................. 67
core (runtime model attribute) ..................... 75
CORE_ (predefined symbol) ......................... 242
eplusplus (predefined symbol) ..................... 242
data_model (runtime model attribute) .......... 75
DATA_MODEL_ (predefined symbol) ............... 242
data16 (extended keyword) ......................... 195, 206
data16_read_addr (intrinsic function) ........... 234
data16_size_t ......................................... 119
data20 (extended keyword) ......................... 195, 206
data20_read_char (intrinsic function) .......... 235
data20_read_long (intrinsic function) ......... 235
data20_read_short (intrinsic function) ........ 235
data20_size_t ......................................... 119
data20_write_char (intrinsic function) .......... 236
data20_write_long (intrinsic function) ....... 236
data20_write_short (intrinsic function) ...... 236
DATE_ (predefined symbol) ......................... 242
delay_cycles (intrinsic function) ............... 236
disable_interrupt (intrinsic function) .......... 236
_DLID_FILE_DESCRIPTOR (configuration symbol) . 66
for creating skeleton code .......................... 89
-O (compiler option) .............................. 178
-o (compiler option) .............................. 179
-Q (linker option) ................................. 43
-r (compiler option) .............................. 162
--char_is_signed (compiler option) ............. 159
--char_is_unsigned (compiler option) .......... 159
--clib (compiler option) ......................... 160
--core (compiler option) ......................... 160
--c89 (compiler option) .......................... 159
--data_model (compiler option) .................. 161
--debug (compiler option) ......................... 162
--dependencies (compiler option) ................ 162
--diagnostics_tables (compiler option) ......... 164
--diag_error (compiler option) ................... 163
--diag_remark (compiler option) ................. 163
--diag_suppress (compiler option) ............... 164
--diag_warning (compiler option) ................. 164
--discard_unused_publics (compiler option) ... 165
--dlib (compiler option) ......................... 165
--dlib_config (compiler option) ................. 165
--double (compiler option) ....................... 166
--ec++ (compiler option) ......................... 167
--ecc++ (compiler option) ......................... 167
--enable_multibyes (compiler option) .......... 168
--error_limit (compiler option) .................. 168
--header_context (compiler option) ............... 169
--library_module (compiler option) ............... 170
--lock_r4 (compiler option) ....................... 171
--lock_r5 (compiler option) ....................... 171
--mfc (compiler option) .......................... 171
--migration_preprocessor_extensions (compiler option) .. 172
--misrac (compiler option) ....................... 157
--misrac_conf (compiler option) ................. 157
--misrac1998 (compiler option) .................. 157
--misrac2004 (compiler option) .................. 157
--module_name (compiler option) .................. 172
--multiplier (compiler option) .................... 173
--multiplier_location (compiler option) ......... 173
--no_code_motion (compiler option) .............. 174
--no_cse (compiler option) ....................... 174
--no_inline (compiler option) .................... 174
--no_path_in_file_macros (compiler option) .... 175
--no_static_destruction (compiler option) ...... 175
--no_system_include (compiler option) .......... 175
--no_typeinfo (compiler option) ................. 176
--no_unroll (compiler option) .................... 177
--no_warnings (compiler option) ................. 177
--preinclude (compiler option) ................... 180
--preprocess (compiler option) ................... 180
--reduce_stack_usage (compiler option) .......... 181
--regvar_r4 (compiler option) .................... 181
--regvar_r5 (compiler option) .................... 182
--relaxed_fp (compiler option) ................... 182
--remsg (compiler option) ......................... 183
--require_prototypes (compiler option) .......... 183
--save_reg20 (compiler option) ................... 183
--segment (compiler option) ....................... 184
--silent (compiler option) ....................... 185
--strict (compiler option) ....................... 185
--system_include_dir (compiler option) .......... 186
--vla (compiler option) .......................... 186
--warnings_affect_exit_code (compiler option) .. 150, 186
--warnings_are_errors (compiler option) ......... 187
?C_EXIT (assembler label) ......................... 83
?C_GETCHAR (assembler label) .................... 83
?C_PUTCHAR (assembler label) .................... 83
@ (operator) placing at absolute address .......... 132
placing in segments .............................. 133
#include files, specifying ......................... 148, 169
#warning message (preprocessor extension) ............ 245
%Z replacement string, implementation-defined behavior ............ 282

**Numerics**

32-bits (floating-point format) ................. 194
64-bit data types, avoiding ......................... 127
64-bits (floating-point format) ..................... 194