

IAR Embedded Workbench[®]

IAR C/C++ Compiler User Guide

for the Texas Instruments

MSP430 Microcontroller Family



COPYRIGHT NOTICE

© 1996–2016 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, C-RUN, C-STAT, visualSTATE, Focus on Your Code, IAR KickStart Kit, IAR Experiment!, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Texas Instruments is a registered trademark of Texas Instruments Corporation. MSP430 is a trademark of Texas Instruments Corporation.

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

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

EDITION NOTICE

Twelfth edition: June 2016

Part number: C430-12

This guide applies to version 6.x of IAR Embedded Workbench® for the Texas Instruments MSP430 microcontroller family.

Internal reference: M20, Hom7.4, docsrc-6.10, V_140324, IMAE.

Brief contents

Tables	27
Preface	29
Part I. Using the compiler	37
Introduction to the IAR build tools	39
Developing embedded applications	45
Data storage	59
Functions	71
Linking overview	85
Linking your application	101
The DLIB runtime environment	115
The CLIB runtime environment	159
Assembler language interface	167
Using C	193
Using C++	201
Application-related considerations	217
Efficient coding for embedded applications	227
Part 2. Reference information	247
External interface details	249
Compiler options	255
Data representation	295
Extended keywords	309

Pragma directives	323
Intrinsic functions	347
The preprocessor	361
C/C++ standard library functions	369
Segment reference	379
The stack usage control file	397
Implementation-defined behavior for Standard C	405
Implementation-defined behavior for C89	421
Index	435

Contents

Tables	27
Preface	29
Who should read this guide	29
Required knowledge	29
How to use this guide	29
What this guide contains	30
Part 1. Using the compiler	30
Part 2. Reference information	31
Other documentation	31
User and reference guides	32
The online help system	32
Further reading	33
Web sites	33
Document conventions	34
Typographic conventions	34
Naming conventions	35
Part I. Using the compiler	37
Introduction to the IAR build tools	39
The IAR build tools—an overview	39
IAR C/C++ Compiler	39
IAR Assembler	39
The IAR XLINK Linker	40
External tools	40
IAR language overview	40
Device support	41
Supported MSP430 devices	41
Preconfigured support files	41
Examples for getting started	42

Special support for embedded systems	42
Extended keywords	42
Pragma directives	43
Predefined symbols	43
Accessing low-level features	43
Developing embedded applications	45
Developing embedded software using IAR build tools	45
CPU features and constraints	45
Mapping of memory	45
Communication with peripheral units	46
Event handling	46
System startup	46
Real-time operating systems	47
The build process—an overview	47
The translation process	47
The linking process	48
After linking	49
Application execution—an overview	50
The initialization phase	50
The execution phase	53
The termination phase	53
Building applications—an overview	54
Basic project configuration	54
Processor configuration	55
Data model (MSP430X only)	56
Code model (MSP430X only)	56
Size of double floating-point type	57
Optimization for speed and size	57
Normal or position-independent code	57
Data storage	59
Introduction	59
Different ways to store data	60

Memory types (MSP430X only)	60
Introduction to memory types	60
Using data memory attributes	61
Pointers and memory types	63
Structures and memory types	64
More examples	64
C++ and memory types	65
Data models	66
Specifying a data model	66
Storage of auto variables and parameters	68
The stack	68
Dynamic memory on the heap	69
Potential problems	70
Functions	71
Function-related extensions	71
Code models (MSP430X only)	71
The Small code model	72
The Large code model	72
Comparing the code models	72
Primitives for interrupts, concurrency, and OS-related programming	73
Interrupt functions	73
Monitor functions	76
Execution in RAM	79
Position-independent code and read-only data	80
Drawbacks and limitations	80
Initialization of pointers	81
Non-ROPI code in ROPI systems	81
Inlining functions	81
C versus C++ semantics	82
Features controlling function inlining	82

Linking overview	85
Linking—an overview	85
Segments and memory	86
What is a segment?	86
The linking process in detail	87
Placing code and data—the linker configuration file	88
The contents of the linker configuration file	89
Initialization at system startup	89
Static data memory segments	90
The initialization process	91
Stack usage analysis	91
Introduction to stack usage analysis	92
Performing a stack usage analysis	92
Result of an analysis—the map file contents	93
Specifying additional stack usage information	95
Limitations	96
Situations where warnings are issued	97
Call graph log	97
Linking your application	101
Linking considerations	101
Placing segments	102
Placing data	104
Setting up stack memory	105
Setting up heap memory	106
Placing code	107
Keeping modules	108
Keeping symbols and segments	109
Application startup	109
Interaction between XLINK and your application	109
Producing other output formats than UBROF	110
Verifying the linked result of code and data placement	110
Segment too long errors and range errors	110
Linker map file	110

Checking module consistency	111
Runtime model attributes	111
Using runtime model attributes	112
Predefined runtime attributes	113
The DLIB runtime environment	115
Introduction to the runtime environment	115
Runtime environment functionality	115
Briefly about input and output (I/O)	116
Briefly about C-SPY emulated I/O	118
Briefly about retargeting	118
Setting up the runtime environment	119
Setting up your runtime environment	120
Retargeting—Adapting for your target system	122
Overriding library modules	123
Customizing and building your own runtime library	124
Additional information on the runtime environment	126
Runtime library configurations	126
Prebuilt runtime libraries	127
Formatters for printf	129
Formatters for scanf	130
The C-SPY emulated I/O mechanism	131
Math functions	131
The Texas Instruments MathLib library	134
System startup and termination	135
System initialization	138
The DLIB low-level I/O interface	139
abort	140
clock	140
__close	140
__exit	141
getenv	141
__getzone	142
__lseek	142

__open	142
raise	143
__read	143
remove	143
rename	144
_ReportAssert	144
signal	144
system	145
__time32, __time64	145
__write	145
Configuration symbols for printf and scanf	146
Configuration symbols for file input and output	147
Locale	147
Strtod	150
Hardware multiplier support	150
MPU/IPE support	150
Managing a multithreaded environment	152
Multithread support in the DLIB runtime environment	153
Enabling multithread support	153
TLS in the linker configuration file	158
The CLIB runtime environment	159
Using a prebuilt runtime library	159
Choosing a runtime library	160
Runtime library filename syntax	160
Input and output	160
Character-based I/O	160
Formatters used by printf and sprintf	161
Formatters used by scanf and sscanf	162
System startup and termination	163
System startup	163
System termination	164
Overriding default library modules	164
Customizing system initialization	164

C-SPY emulated I/O	164
The debugger Terminal I/O window	165
Termination	165
Hardware multiplier support	165
MPU/IPE support	165
Assembler language interface	167
Mixing C and assembler	167
Intrinsic functions	167
Mixing C and assembler modules	168
Inline assembler	169
Reference information for inline assembler	170
An example of how to use clobbered memory	175
Calling assembler routines from C	175
Creating skeleton code	176
Compiling the skeleton code	176
Calling assembler routines from C++	178
Calling convention	179
Choosing a calling convention	179
Function declarations	180
Using C linkage in C++ source code	180
Preserved versus scratch registers	181
Function entrance	182
Function exit	183
Restrictions for special function types	184
Examples	184
Assembler instructions used for calling functions	186
Normal (non-ROPI) code	186
Position-independent code and read-only data	186
Interrupt functions	187
Module consistency	187
Memory access methods	187
The data16 memory access method	188
The data20 memory access method	188

Call frame information	189
CFI directives	189
Creating assembler source with CFI support	190
Using C	193
C language overview	193
Extensions overview	194
Enabling language extensions	195
IAR C language extensions	195
Extensions for embedded systems programming	196
Relaxations to Standard C	198
Using C++	201
Overview—EC++ and EEC++	201
Embedded C++	201
Extended Embedded C++	202
Enabling support for C++	203
EC++ feature descriptions	203
Using IAR attributes with Classes	203
Function types	206
New and Delete operators	207
Using static class objects in interrupts	208
Using New handlers	209
Templates	209
Debug support in C-SPY	209
EEC++ feature description	209
Templates	210
Variants of cast operators	213
Mutable	214
Namespace	214
The STD namespace	214
C++ language extensions	214

Application-related considerations	217
Stack considerations	217
Stack size considerations	217
Heap considerations	217
Heap segments in DLIB	218
Heap segments in CLIB	218
Heap size and standard I/O	218
Interaction between the tools and your application	219
Checksum calculation for verifying image integrity	220
Briefly about checksum calculation	220
Calculating and verifying a checksum	222
Troubleshooting checksum calculation	226
Efficient coding for embedded applications	227
Selecting data types	227
Using efficient data types	227
Floating-point types	227
Alignment of elements in a structure	228
Anonymous structs and unions	229
Controlling data and function placement in memory	230
Data placement at an absolute location	231
Data and function placement in segments	233
Controlling compiler optimizations	234
Scope for performed optimizations	234
Multi-file compilation units	235
Optimization levels	236
Speed versus size	236
Fine-tuning enabled transformations	237
Facilitating good code generation	239
Writing optimization-friendly source code	240
Saving stack space and RAM memory	240
Function prototypes	240
Integer types and bit negation	241
Protecting simultaneously accessed variables	242

Accessing special function registers	243
Non-initialized variables	245
Efficient switch statements	245
Part 2. Reference information	247
External interface details	249
Invocation syntax	249
Compiler invocation syntax	249
Passing options	250
Environment variables	250
Include file search procedure	250
Compiler output	251
Error return codes	252
Diagnostics	253
Message format	253
Severity levels	253
Setting the severity level	254
Internal error	254
Compiler options	255
Options syntax	255
Types of options	255
Rules for specifying parameters	255
Summary of compiler options	257
Descriptions of compiler options	261
--c89	261
--char_is_signed	262
--char_is_unsigned	262
--clib	262
--code_model	262
--core	263
-D	263
--data_model	264

--debug, -r	265
--dependencies	265
--diag_error	266
--diag_remark	266
--diag_suppress	267
--diag_warning	267
--diagnostics_tables	268
--discard_unused_publics	268
--dlib	268
--dlib_config	269
--double	270
-e	270
--ec++	270
--eec++	271
--enable_multibytes	271
--enable_restrict	271
--error_limit	272
-f	272
--guard_calls	272
--header_context	273
--hw_workaround	273
-I	274
-l	274
--library_module	275
--lock_r4	275
--lock_r5	275
--macro_positions_in_diagnostics	276
--mfc	276
--migration_preprocessor_extensions	276
--module_name	277
--multiplier	277
--no_call_frame_info	278
--no_code_motion	278
--no_cse	278

--no_inline	279
--no_path_in_file_macros	279
--no_rw_dynamic_init	279
--no_size_constraints	280
--no_static_destruction	280
--no_system_include	280
--no_tbaa	281
--no_typedefs_in_diagnostics	281
--no_ubrof_messages	282
--no_unroll	282
--no_warnings	282
--no_wrap_diagnostics	282
-O	283
--omit_types	283
--only_stdout	284
--output, -o	284
--pending_instantiations	284
--predef_macros	285
--preinclude	285
--preprocess	285
--public_equ	286
--reduce_stack_usage	286
--regvar_r4	286
--regvar_r5	287
--relaxed_fp	287
--remarks	288
--require_prototypes	288
--ropi	288
--save_reg20	289
--segment	289
--silent	290
--strict	290
--system_include_dir	291
--use_cplusplus_inline	291

--version	291
--vla	292
--warn_about_c_style_casts	292
--warnings_affect_exit_code	292
--warnings_are_errors	292
Data representation	295
Alignment	295
Alignment on the MSP430 microcontroller	296
Basic data types—integer types	296
Integer types—an overview	296
Bool	297
The long long type	297
The enum type	297
The char type	297
The wchar_t type	297
Bitfields	298
Basic data types—floating-point types	299
Floating-point environment	299
32-bit floating-point format	300
64-bit floating-point format	300
Representation of special floating-point numbers	300
Pointer types	301
Function pointers	301
Data pointers	301
Casting	302
Structure types	303
Alignment of structure types	303
General layout	303
Packed structure types	303
Type qualifiers	305
Declaring objects volatile	305
Declaring objects volatile and const	306
Declaring objects const	307

Data types in C++	307
Extended keywords	309
General syntax rules for extended keywords	309
Type attributes	309
Object attributes	311
Summary of extended keywords	312
Descriptions of extended keywords	313
__cc_version1	313
__cc_version2	314
__data16	314
__data20	314
__interrupt	315
__intrinsic	315
__monitor	315
__no_alloc, __no_alloc16	316
__no_alloc_str, __no_alloc_str16	316
__no_init	317
__no_multiplier	317
__no_pic	318
__noreturn	318
__persistent	318
__ramfunc	319
__raw	319
__regvar	319
__root	320
__ro_placement	320
__save_reg20	321
__task	321
Pragma directives	323
Summary of pragma directives	323
Descriptions of pragma directives	325
basic_template_matching	325
bis_nmi_ie1	325

bitfields	326
calls	327
call_graph_root	327
constseg	328
data_alignment	328
dataseg	329
default_function_attributes	329
default_variable_attributes	330
diag_default	331
diag_error	331
diag_remark	332
diag_suppress	332
diag_warning	332
error	333
include_alias	333
inline	334
language	334
location	335
message	336
no_epilogue	336
object_attribute	337
optimize	337
pack	339
public_equ	339
__printf_args	340
required	340
rtmodel	341
__scanf_args	341
segment	342
STDC CX_LIMITED_RANGE	343
STDC FENV_ACCESS	343
STDC FP_CONTRACT	344
type_attribute	344
unroll	345

vector	345
weak	346
Intrinsic functions	347
Summary of intrinsic functions	347
Descriptions of intrinsic functions	349
__bcd_add_type	349
__bic_SR_register	349
__bic_SR_register_on_exit	350
__bis_GIE_interrupt_state	350
__bis_SR_register	350
__bis_SR_register_on_exit	350
__code_distance	350
__data16_read_addr	351
__data16_write_addr	351
__data20_longjmp	352
__data20_read_type	352
__data20_setjmp	352
__data20_write_type	353
__delay_cycles	353
__disable_interrupt	353
__enable_interrupt	354
__even_in_range	354
__get_interrupt_state	354
__get_R4_register	355
__get_R5_register	355
__get_SP_register	355
__get_SR_register	356
__get_SR_register_on_exit	356
__low_power_mode_n	356
__low_power_mode_off_on_exit	356
__no_operation	356
__op_code	357
__saturated_add_signed_type	357

__saturated_add_unsigned_type	357
__saturated_sub_signed_type	357
__saturated_sub_unsigned_type	358
__set_interrupt_state	358
__set_R4_register	358
__set_R5_register	358
__set_SP_register	359
__swap_bytes	359
The preprocessor	361
Overview of the preprocessor	361
Description of predefined preprocessor symbols	362
__BASE_FILE__	362
__BUILD_NUMBER__	362
__CODE_MODEL__	362
__CORE__	362
__COUNTER__	362
__cplusplus	362
__DATA_MODEL__	363
__DATE__	363
__embedded_cplusplus	363
__FILE__	363
__func__	364
__FUNCTION__	364
__IAR_SYSTEMS_ICC__	364
__ICC_430__	364
__LINE__	364
__POSITION_INDEPENDENT_CODE__	364
__PRETTY_FUNCTION__	365
__REGISTER_MODEL__	365
__REGISTER_R4__	365
__REGISTER_R5__	365
__ROPI__	365
__STDC__	365

__STDC_VERSION__	366
__SUBVERSION__	366
__TIME__	366
__TIMESTAMP__	366
__VER__	366
Descriptions of miscellaneous preprocessor extensions	366
NDEBUG	367
#warning message	367
C/C++ standard library functions	369
C/C++ standard library overview	369
Header files	370
Library object files	370
Alternative more accurate library functions	370
Reentrancy	370
The longjmp function	371
DLIB runtime environment—implementation details	371
C header files	372
C++ header files	373
Library functions as intrinsic functions	375
Added C functionality	375
Symbols used internally by the library	377
CLIB runtime environment—implementation details	378
Library definitions summary	378
Segment reference	379
Summary of segments	379
Descriptions of segments	381
BSLSIGNATURE	381
CHECKSUM	381
CODE	382
CODE_I	382
CODE_ID	382
CODE_PAD	383
CODE16	383

CSTACK	383
CSTART	384
DATA16_AC	384
DATA16_AN	384
DATA16_C	384
DATA16_HEAP	385
DATA16_I	385
DATA16_ID	385
DATA16_N	386
DATA16_P	386
DATA16_Z	386
DATA20_AC	387
DATA20_AN	387
DATA20_C	387
DATA20_HEAP	387
DATA20_I	388
DATA20_ID	388
DATA20_N	389
DATA20_P	389
DATA20_Z	389
DIFUNCT	390
INFO	390
INFOA	390
INFOB	390
INFOC	391
INFOD	391
INTVEC	391
IPE_B1	391
IPE_B2	392
IPECODE16	392
IPEDATA16_C	392
IPEDATA16_N	393
IPESIGNATURE	393
ISR_CODE	393

JTAGSIGNATURE	393
MPU_B1	394
MPU_B2	394
REGVAR_AN	394
RESET	394
SIGNATURE	395
TLS16_I	395
TLS16_ID	395
The stack usage control file	397
Overview	397
C++ names	397
Stack usage control directives	397
call graph root directive	397
check that directive	398
exclude directive	398
function directive	399
max recursion depth directive	399
no calls from directive	400
possible calls directive	400
Syntactic components	401
<i>category</i>	401
<i>func-spec</i>	401
<i>module-spec</i>	402
<i>name</i>	402
<i>call-info</i>	402
<i>stack-size</i>	403
<i>size</i>	403
Implementation-defined behavior for Standard C	405
Descriptions of implementation-defined behavior	405
J.3.1 Translation	405
J.3.2 Environment	406
J.3.3 Identifiers	407
J.3.4 Characters	407

J.3.5 Integers	409
J.3.6 Floating point	409
J.3.7 Arrays and pointers	410
J.3.8 Hints	411
J.3.9 Structures, unions, enumerations, and bitfields	411
J.3.10 Qualifiers	412
J.3.11 Preprocessing directives	412
J.3.12 Library functions	414
J.3.13 Architecture	418
J.4 Locale	419
Implementation-defined behavior for C89	421
Descriptions of implementation-defined behavior	421
Translation	421
Environment	421
Identifiers	422
Characters	422
Integers	423
Floating point	424
Arrays and pointers	425
Registers	425
Structures, unions, enumerations, and bitfields	425
Qualifiers	426
Declarators	426
Statements	426
Preprocessing directives	426
Library functions for the IAR DLIB Runtime Environment	428
Library functions for the CLIB runtime environment	431
Index	435

Tables

1: Typographic conventions used in this guide	34
2: Naming conventions used in this guide	35
3: Memory types and their corresponding memory attributes	61
4: Data model characteristics	66
5: Heaps supported in memory types	70
6: Code models	72
7: segments holding initialized data	90
8: Example of runtime model attributes	112
9: Predefined runtime model attributes	113
10: Debug information and levels of C-SPY emulated I/O	120
11: Library configurations	127
12: Formatters for printf	129
13: Formatters for scanf	130
14: Extended command line files for the small math functions	132
15: DLIB low-level I/O interface functions	139
16: Descriptions of printf configuration symbols	146
17: Descriptions of scanf configuration symbols	147
18: Additional linker configuration files for the hardware multiplier	150
19: Library objects using TLS	153
20: Macros for implementing TLS allocation	156
21: Inline assembler operand constraints	172
22: Supported constraint modifiers	172
23: List of valid clobbers	174
24: Operand modifiers and transformations	175
25: Registers used for passing parameters	182
26: Registers used for returning values	184
27: Call frame information resources defined in a names block	190
28: Language extensions	195
29: Compiler optimization levels	236
30: Compiler environment variables	250
31: Error return codes	252

32: Compiler options summary	257
33: Integer types	296
34: Floating-point types	299
35: Function pointers	301
36: Data pointers	301
37: Extended keywords summary	312
38: Pragma directives summary	323
39: Intrinsic functions summary	347
40: Functions for binary coded decimal operations	349
41: Functions for reading data that has a 20-bit address	352
42: Functions for writing data that has a 20-bit address	353
43: Traditional Standard C header files—DLIB	372
44: C++ header files	373
45: Standard template library header files	374
46: New Standard C header files—DLIB	374
47: CLIB runtime environment header files	378
48: Segment summary	379
49: Message returned by <code>strerror()</code> —DLIB runtime environment	420
50: Message returned by <code>strerror()</code> —DLIB runtime environment	431
51: Message returned by <code>strerror()</code> —CLIB runtime environment	434

Preface

Welcome to the *IAR C/C++ Compiler User 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.

REQUIRED KNOWLEDGE

To use the tools in IAR Embedded Workbench, you should have working knowledge of:

- The architecture and instruction set of the MSP430 microcontroller (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 31.

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 this product, we suggest that you first read the guide *Getting Started with IAR Embedded Workbench®* for an overview of the tools and the features that the IDE offers. The tutorials, which you can find in IAR Information Center, will help you get started using IAR Embedded Workbench.

What this guide contains

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

PART I. USING THE COMPILER

- *Introduction to the IAR build tools* gives an introduction to the IAR build tools, which includes an overview of the tools, the programming languages, the available device support, and extensions provided for supporting specific features of the MSP430 microcontroller.
- *Developing embedded applications* gives the information you need to get started developing your embedded software using the IAR build tools.
- *Data storage* describes how to store data in memory.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Linking overview* describes the linking process using the IAR XLINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using XLINK options and tailoring the linker configuration file.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *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++.
- *Application-related considerations* discusses a selected range of application issues related to using the compiler and linker.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

PART 2. REFERENCE INFORMATION

- *External interface details* provides reference information about how the compiler 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.
- *C/C++ standard 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.
- *The stack usage control file* describes the syntax and semantics of stack usage control files.
- *Implementation-defined behavior for Standard C* describes how the compiler handles the implementation-defined areas of Standard C.
- *Implementation-defined behavior for C89* describes how the compiler handles the implementation-defined areas of the C language standard C89.

Other documentation

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

USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for MSP430*.
- Programming for the IAR C/C++ Compiler for MSP430, is available in the *IAR C/C++ Compiler User Guide for MSP430*.
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, is available in the *IAR Linker and Library Tools Reference Guide*.
- Programming for the IAR Assembler for MSP430, is available in the *IAR Assembler User Guide for MSP430*.
- Using the IAR DLIB C/C++ standard library functions, is available in the *C/C++ standard library reference*, available in the online help system.
- Using the IAR CLIB C standard library, is available in the *IAR C Library Functions Reference Guide*, available in the online help system.
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for MSP430, is available in the *IAR Embedded Workbench® Migration Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about project management, editing, and building in the IDE
- Information about debugging using the IAR C-SPY® Debugger

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

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.
- Meyers, Scott. *Effective STL*. Addison-Wesley.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++*. Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.

WEB SITES

Recommended web sites:

- The Texas Instruments web site, www.ti.com, that contains information and news about the MSP430 microcontrollers.

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

Document conventions

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

When referring to a directory in your product installation, for example `430\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\430\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> • Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
<code>parameter</code>	A placeholder for an actual value used as a parameter, for example <code>filename.h</code> where <code>filename</code> represents the name of the file.
<code>[option]</code>	An optional part of a directive, where <code>[</code> and <code>]</code> are not part of the actual directive, but any <code>,</code> <code>,</code> <code>{</code> , or <code>}</code> are part of the directive syntax.
<code>{option}</code>	A mandatory part of a directive, where <code>{</code> and <code>}</code> are not part of the actual directive, but any <code>,</code> <code>,</code> <code>{</code> , or <code>}</code> are part of the directive syntax.
<code>[option]</code>	An optional part of a command.
<code>[a b c]</code>	An optional part of a command with alternatives.
<code>{a b c}</code>	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.

Table 1: Typographic conventions used in this guide





Style	Used for
<i>italic</i>	<ul style="list-style-type: none"> • A cross-reference within this guide or to another guide. • Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

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

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

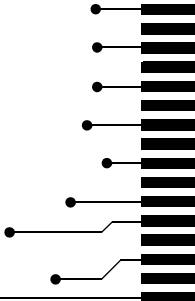
Brand name	Generic term
IAR Embedded Workbench® for MSP430	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for MSP430	the IDE
IAR C-SPY® Debugger for MSP430	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for MSP430	the compiler
IAR Assembler™ for MSP430	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Runtime Environment™	the DLIB runtime environment
IAR CLIB Runtime Environment™	the CLIB runtime environment

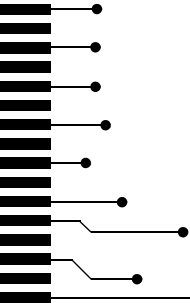
Table 2: Naming conventions used in this guide

Part I. Using the compiler

This part of the *IAR C/C++ Compiler User Guide for MSP430* includes these chapters:

- Introduction to the IAR build tools
- Developing embedded applications
- Data storage
- Functions
- Linking overview
- Linking your application
- The DLIB runtime environment
- The CLIB runtime environment
- Assembler language interface
- Using C
- Using C++
- Application-related considerations
- Efficient coding for embedded applications.





Introduction to the IAR build tools

- The IAR build tools—an overview
- IAR language overview
- Device support
- Special support for embedded systems

The IAR build tools—an overview

In the IAR product installation you can find a set of tools, code examples, and user documentation, all suitable for developing software for MSP430-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.



IAR Embedded Workbench® is a very powerful Integrated Development Environment (IDE) that allows you to develop and manage complete embedded application projects. It provides an easy-to-learn and highly efficient development environment with maximum code inheritance capabilities, comprehensive and specific target support. IAR Embedded Workbench promotes a useful working methodology, and thus a significant reduction of the development time.

For information about the IDE, see the *IDE Project Management and Building Guide*.



The compiler, assembler, and linker can also be run from a command line environment, if you want to use them as external tools in an already established project environment.

IAR C/C++ COMPILER

The IAR C/C++ Compiler for MSP430 is a state-of-the-art compiler that offers the standard features of the C and C++ languages, plus extensions designed to take advantage of the MSP430-specific facilities.

IAR ASSEMBLER

The IAR Assembler for MSP430 is a powerful relocating macro assembler with a versatile set of directives and expression operators. The assembler features a built-in C language preprocessor and supports conditional assembly.

The IAR Assembler for MSP430 uses the same mnemonics and operand syntax as the Texas Instruments MSP430 Assembler, which simplifies the migration of existing code. For more information, see the *IAR Assembler User Guide for MSP430*.

THE IAR XLINK LINKER

The IAR XLINK Linker is a powerful, flexible software tool for use in the development of embedded controller applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable input, multi-module, C/C++, or mixed C/C++ and assembler programs.

To handle libraries, the library tools XAR and XLIB are included.

EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IDE Project Management and Building Guide*.

IAR language overview

The IAR C/C++ Compiler for MSP430 supports:

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

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

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

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

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

Device support

To get a smooth start with your product development, the IAR product installation comes with a wide range of device-specific support.

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.

PRECONFIGURED SUPPORT FILES

The IAR product installation contains preconfigured files for supporting different devices. If you need additional files for device support, they can be created using one of the provided ones as a template.

Header files for I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension `.h`. The product package supplies I/O files for all devices that are available at the time of the product release. You can find these files in the `430\inc` directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can be created using one of the provided ones as a template.

There are two types of I/O header files:

- Files named `iodevice.h` (for example `io430x44x.h` and `io430fr5969.h`) use C unions and structures with bitfields to describe special function registers and the individual fields.

- Files named `mspdevice.h` (for example `msp430f449.h` and `cc430f5123.h`) use the Texas Instruments style definitions, where special function registers are defined using plain integer types and individual bits are defined using preprocessor macros.

The generic I/O header files `io430.h` and `mcp430.h` can be used for including the correct I/O header file for the device that is used. A preprocessor symbol on the form `__DEVICE__` (for example `__MSP430SL5438A__`) must be defined. When used in IAR Embedded Workbench, this preprocessor symbol is automatically defined.

For more information, see *Accessing special function registers*, page 243.

Linker configuration files

The `430\config` directory contains ready-made linker configuration files for all supported devices. The files have the filename extension `.xcl` and contain the information required by the linker. For more information about the linker configuration file, see *Placing code and data—the linker configuration file*, page 88 as well as the *IAR Linker and Library Tools Reference Guide*.

Device description files

The debugger handles several of the device-specific requirements, such as definitions of available memory areas, peripheral registers and groups of these, by using device description files. These files are located in the `430\config` directory and they have the filename extension `.ddf`. The peripheral registers and groups of these can be defined in separate files (filename extension `.sfr`), which in that case are included in the `.ddf` file. For more information about these files, see the *C-SPY® Debugging Guide for MSP430*.

EXAMPLES FOR GETTING STARTED

The `430\examples` directory contains several examples of working applications to give you a smooth start with your development. Examples are provided for most of the supported device families.

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 how to access and store data objects, as well as for controlling how a function should work internally and how it should be called/returned.

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 270 for additional information.

For more information about the extended keywords, see the chapter *Extended keywords*. See also, *Data storage*, page 59 and *Functions*, page 71.

PRAGMA DIRECTIVES

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

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

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

PREDEFINED SYMBOLS

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example time of compilation or the build number of the compiler.

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

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 167.

Developing embedded applications

- Developing embedded software using IAR build tools
- The build process—an overview
- Application execution—an overview
- Building applications—an overview
- Basic project configuration

Developing embedded software using IAR build tools

Typically, embedded software written for a dedicated microcontroller is designed as an endless loop waiting for some external events to happen. The software is located in ROM and executes on reset. You must consider several hardware and software factors when you write this kind of software. To your help, you have compiler options, extended keywords, pragma directives, etc.

CPU FEATURES AND CONSTRAINTS

Some of the basic features of the MSP430 microcontroller is ultra low-power mode support and a wide range of devices targeted at specific kinds of embedded systems.

When developing software for the MSP430 microcontroller, you must consider some CPU constraints, such as the instruction set and features like hardware multiplier, memory protection unit, and memory configuration.

The compiler supports this by means of compiler options, extended keywords, pragma directives, etc.

MAPPING OF MEMORY

Embedded systems typically contain various types of memory, such as RAM, FRAM, ROM, EEPROM, or flash memory.

As an embedded software developer, you must understand the features of the different types of memory. For example, on-chip RAM is often faster than other types of memories, and variables that are accessed often would in time-critical applications

benefit from being placed here. Conversely, some configuration data might be accessed seldom but must maintain their value after power off, so they should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. For more information, see Controlling data and function placement in memory, page 230. The linker places sections of code and data in memory according to the directives you specify in the linker configuration file, see *Placing code and data—the linker configuration file*, page 88.

COMMUNICATION WITH PERIPHERAL UNITS

If external devices are connected to the microcontroller, you might need to initialize and control the signaling interface, for example by using chip select pins, and detect and handle external interrupt signals. Typically, this must be initialized and controlled at runtime. The normal way to do this is to use special function registers (SFR). These are typically available at dedicated addresses, containing bits that control the chip configuration.

Standard peripheral units are defined in device-specific I/O header files with the filename extension `.h`. See *Device support*, page 41. For an example, see Accessing special function registers, page 243.

EVENT HANDLING

In embedded systems, using *interrupts* is a method for handling external events immediately; for example, detecting that a button was pressed. In general, when an interrupt occurs in the code, the microcontroller immediately stops executing the code it runs, and starts executing an interrupt routine instead.

The compiler provides various primitives for managing hardware and software interrupts, which means that you can write your interrupt routines in C, see Primitives for interrupts, concurrency, and OS-related programming, page 73.

SYSTEM STARTUP

In all embedded systems, system startup code is executed to initialize the system—both the hardware and the software system—before the `main` function of the application is called.

As an embedded software developer, you must ensure that the startup code is located at the dedicated memory addresses, or can be accessed using a pointer from the vector table. This means that startup code and the initial vector table must be placed in non-volatile memory, such as ROM, EEPROM, or flash.

A C/C++ application further needs to initialize all global variables. This initialization is handled by the linker and the system startup code in conjunction. For more information, see *Application execution—an overview*, page 50.

REAL-TIME OPERATING SYSTEMS

In many cases, the embedded application is the only software running in the system. However, using an RTOS has some advantages.

For example, the timing of high-priority tasks is not affected by other parts of the program which are executed in lower priority tasks. This typically makes a program more deterministic and can reduce power consumption by using the CPU efficiently and putting the CPU in a lower-power state when idle.

Using an RTOS can make your program easier to read and maintain, and in many cases smaller as well. Application code can be cleanly separated in tasks which are truly independent of each other. This makes teamwork easier, as the development work can be easily split into separate tasks which are handled by one developer or a group of developers.

Finally, using an RTOS reduces the hardware dependence and creates a clean interface to the application, making it easier to port the program to different target hardware.

The build process—an overview

This section gives an overview of the build process; how the various build tools—compiler, assembler, and linker—fit together, going from source code to an executable image.

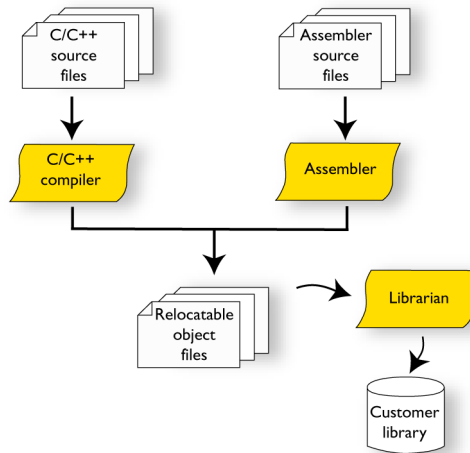
To get familiar with the process in practice, you should run one or more of the tutorials available from the IAR Information Center.

THE TRANSLATION PROCESS

There are two tools in the IDE that translate application source files to intermediary object files. The IAR C/C++ Compiler and the IAR Assembler. Both produce relocatable object files in the IAR UBROF format.

Note: The compiler can also be used for translating C source code into assembler source code. If required, you can modify the assembler source code which then can be assembled into object code. For more information about the IAR Assembler, see the *IAR Assembler User Guide for MSP430*.

This illustration shows the translation process:



After the translation, you can choose to pack any number of modules into an archive, or in other words, a library. The important reason you should use libraries is that each module in a library is conditionally linked in the application, or in other words, is only included in the application if the module is used directly or indirectly by a module supplied as an object file. Optionally, you can create a library; then use the IAR XAR Library Builder or the IAR XLIB Librarian.

THE LINKING PROCESS

The relocatable modules, in object files and libraries, produced by the IAR compiler and assembler cannot be executed as is. To become an executable application, they must be *linked*.

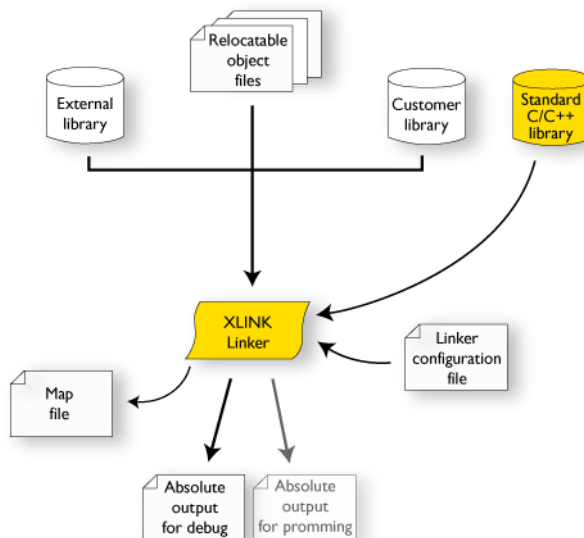
The IAR XLINK Linker (`xlink.exe`) is used for building the final application. Normally, the linker requires the following information as input:

- Several object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label (set by default)
- The linker configuration file that describes placement of code and data in the memory of the target system
- Information about the output format.

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 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.

This illustration shows the linking process:



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

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

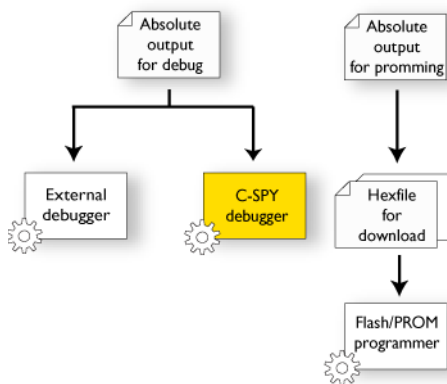
For more information about the procedure performed by the linker, see the *IAR Linker and Library Tools Reference Guide*.

AFTER LINKING

The IAR XLINK Linker produces an absolute object file in the output format you specify. After linking, the produced absolute executable image can be used for:

- Loading into the IAR C-SPY Debugger or any other compatible external debugger that reads UBROF.
- Programming to a flash/PROM using a flash/PROM programmer.

This illustration shows the possible uses of the absolute output files:



Application execution—an overview

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

- Initialization phase
- Execution phase
- Termination phase.

THE INITIALIZATION PHASE

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

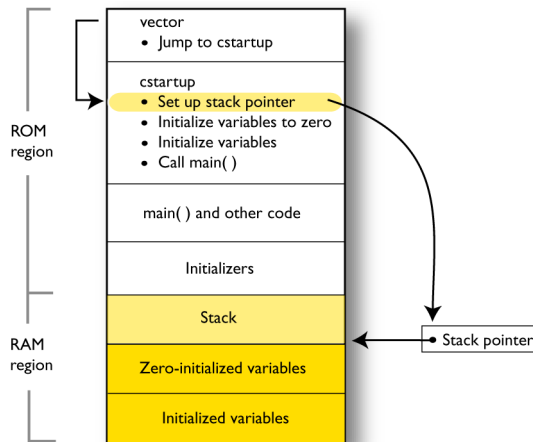
- Hardware initialization, which generally at least initializes the stack pointer.
The hardware initialization is typically performed in the system startup code `cstartup.s43` and if required, by an extra low-level routine that you provide. It might include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.
- Software C/C++ system initialization
Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.
- Application initialization
This depends entirely on your application. It can include setting up an RTOS kernel and starting initial tasks for an RTOS-driven application. For a bare-bone application,

it can include setting up various interrupts, initializing communication, initializing devices, etc.

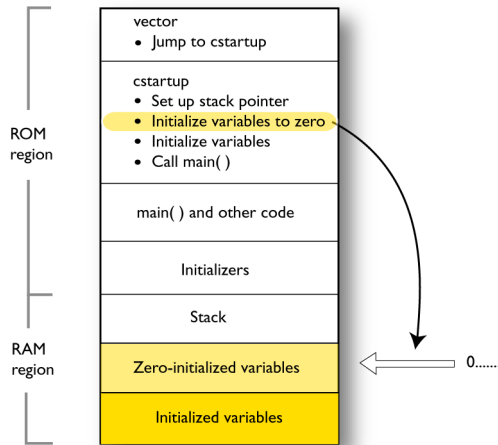
For a ROM/flash-based system, constants and functions are already placed in ROM. All symbols placed in RAM must be initialized before the `main` function is called. The linker has already divided the available RAM into different areas for variables, stack, heap, etc.

The following sequence of illustrations gives a simplified overview of the different stages of the initialization.

- I When an application is started, the system startup code first performs hardware initialization, such as initialization of the stack pointer to point at the end of the predefined stack area:

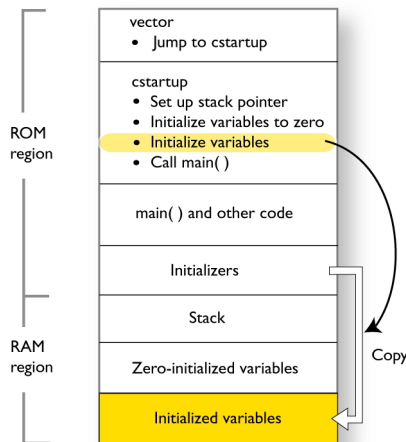


- Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

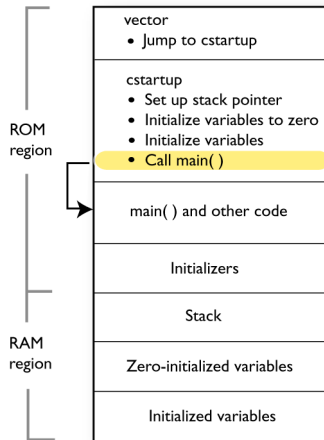


Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`

- For *initialized data*, data declared, for example, like `int i = 6;` the initializers are copied from ROM to RAM:



4 Finally, the `main` function is called:



For more information about each stage, see *System startup and termination*, page 135. For more information about initialization of data, see *Initialization at system startup*, page 89.

THE EXECUTION PHASE

The software of an embedded application is typically implemented as a loop which is either interrupt-driven or uses polling for controlling external interaction or internal events. For an interrupt-driven system, the interrupts are typically initialized at the beginning of the `main` function.

In a system with real-time behavior and where responsiveness is critical, a multi-task system might be required. This means that your application software should be complemented with a real-time operating system. In this case, the RTOS and the different tasks must also be initialized at the beginning of the `main` function.

THE TERMINATION PHASE

Typically, the execution of an embedded application should never end. If it does, you must define a proper end behavior.

To terminate an application in a controlled way, either call one of the Standard C library functions `exit`, `_Exit`, or `abort`, or return from `main`. If you return from `main`, the `exit` function is executed, which means that C++ destructors for static and global variables are called (C++ only) and all open files are closed.

Of course, in case of incorrect program logic, the application might terminate in an uncontrolled and abnormal way—a system crash.

For more information about this, see *System termination*, page 137.

Building applications—an overview

In the command line interface, this 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 54.

On the command line, this line can be used for starting the linker:

```
xlink myfile.r43 myfile2.r43 -o a.d43 -f my_configfile.xcl -r
```

In this example, `myfile.r43` and `myfile2.r43` are object files, and `my_configfile.xcl` is the linker configuration file. The option `-o` specifies the name of the output file. The option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

Note: By default, the label where the application starts is `__program_start`. You can use the `-s` command line option to change this.



When building a project, the IAR Embedded Workbench IDE can produce extensive build information in the **Build** messages window. This information can be useful, for example, as a base for producing batch files for building on the command line. You can copy the information and paste it in a text file. To activate extensive build information, choose **Tools>Options> Messages** and select the option **Show build messages: All**.

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 need to make settings for:

- Processor configuration
- Data model (only for MSP430X)
- Code model (only for MSP430X)
- Size of `double` floating-point type
- Optimization settings

- Runtime environment, see *Setting up the runtime environment*, page 119
- Position-independent code and read-only data
- Customizing the XLINK configuration, see the chapter *Linking your application*.

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

PROCESSOR CONFIGURATION

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

Core

The compiler supports 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.



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



In the IDE, choose **Project>Options>General Options>Target** and choose an appropriate device from the **Device** drop-down list. The core option will then be automatically selected.

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

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>General Options>Target** and choose 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 linker command line with one of these configuration files:

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

For more information, see Hardware multiplier support, page 150.

DATA MODEL (MSP430X ONLY)

In the MSP430X architecture, there is a trade-off in how memory is accessed. You can use 16-bit addressing which is cheaper but can only access the first 64 kBytes of memory, or 20-bit addressing which is often more expensive, but can access the full 1 MByte of memory.

In the compiler, you can control the trade-off between access methods by selecting a data model. These data models are supported:

- The *Small* data model uses only 16-bit addressing, so it can access the first 64 kBytes of memory. The 20-bit registers are treated as 16 bits wide. To save stack space, the most significant 4 bits of the registers are not preserved across function calls, including interrupt handlers. The only way to access the full 1-MByte memory range is to use intrinsic functions.
- The *Medium* data model uses 16-bit addressing by default, but also allows 20-bit addressing, so the full 1 MByte of memory can be accessed. When mandated by the calling convention, the full 20 bits of the registers are preserved across function calls.
- The *Large* data model uses 20-bit addressing by default, and allows 16-bit addressing. The full 1 MByte of memory can be accessed.

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

CODE MODEL (MSP430X ONLY)

The compiler supports code models to control the call and return sequences. These code models are available:

- The *Small* code model uses 16-bit instructions `CALL` and `RET`, and reaches 64 Kbytes of memory. Function pointers are 2 bytes.
- The *Large* code model uses 20-bit instructions `CALLA` and `RETA`, and reaches 1 MB of memory. Function pointers are 4 bytes. This is the default code model.

The main differences between the code models is that the Small code model is slightly faster and requires less runtime stack. However, code placement is restricted to the first 64 Kbytes.

Note: The call and return sequences are different if `--ropi` is used.

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

SIZE OF DOUBLE FLOATING-POINT TYPE

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE 754 format. If you use the compiler option `--double={32|64}`, you can choose whether data declared as `double` should be represented with 32 bits or 64 bits. The data type `float` is always represented using 32 bits.

OPTIMIZATION FOR SPEED AND SIZE

The compiler's optimizer performs, among other things,

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

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

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

NORMAL OR POSITION-INDEPENDENT CODE

Most applications are designed to be placed at a fixed position in memory. However, sometimes it is useful to instead not decide until at runtime where to place the application. By enabling the compiler option `--ropi`, you can make the compiler generate *position-independent code and read-only data* (ROPI). ROPI allows the application image to execute correctly even if it is placed at a different location than where it was linked. For more information, see *Position-independent code and read-only data*, page 80.

Data storage

- Introduction
- Memory types (MSP430X only)
- Data models
- Storage of auto variables and parameters
- Dynamic memory on the heap

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 peripheral units.

Some devices support FRAM, a memory type that can be written to like RAM, but retains its contents after a power-down like traditional read-only memories. FRAM can be partitioned to contain read-only segments and read/write segments.

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. For more information about this, see *Memory types (MSP430X only)*, page 60.



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 or FRAM 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 either in registers or on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid. For more information, see *Storage of auto variables and parameters*, page 68.
- 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 *Memory types (MSP430X only)*, page 60.
- 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 69.

Memory types (MSP430X only)

This section describes the concept of *memory types* used for accessing data by the compiler. For each memory type, the capabilities and limitations are discussed.

INTRODUCTION TO MEMORY TYPES

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

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

To choose a default memory type that your application will use, select a *data model*. However, it is possible to specify—for individual variables or pointers—different memory types. This makes it possible to create an application that can contain a large

amount of data, and at the same time make sure that variables that are used often are placed in memory that can be efficiently accessed.

Below is an overview of the memory types.

Data16

The data16 memory consists of the low 64 Kbytes of data 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 runtime.

Data20

Using the data20 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.

Note: Data20 is not available in the Small data model.

USING DATA MEMORY ATTRIBUTES

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

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

Memory type	Keyword	Address range	Pointer size	Default in data model
Data16	<code>__data16</code>	0x0–0xFFFF	16 bits	Small and Medium
Data20	<code>__data20</code>	0x0–0xFFFFF	32 bits	Large

Table 3: 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 270 for additional information.

For more information about each keyword, see Descriptions of extended keywords, page 313.

Syntax for type attributes used on data objects

Type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__data16 int i;
int __data16 j;
```

Both `i` and `j` are placed in data16 memory.

Unlike `const` and `volatile`, when a type attribute is used before the type specifier in a derived type, the type attribute applies to the object, or typedef itself.

```
int __data16 * p;      /* integer in data16 memory */
int * __data16 p;     /* pointer in data16 memory */
__data16 int * p;     /* variable in data16 memory */
```

The integer pointed to by `p1` is in data16 memory. The variable `p2` is placed in data16 memory, as is the variable `p3`. In the first two cases, the type attribute behaves in the same way as `const` and `volatile` would.

In all cases, if a memory attribute is not specified, an appropriate default memory type is used.

Using a type definition can sometimes make the code clearer:

```
typedef __data16 d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in data16 memory. The variable `q1` can point to such integers.

You can also use the `#pragma type_attributes` directive to specify type attributes for a declaration. The type attributes specified in the pragma directive are applied to the data object or typedef being declared.

```
#pragma type_attribute=__data16
int * q2;
```

The variable `q2` is placed in data16 memory.

For more examples of using memory attributes, see *More examples*, page 64.

Type definitions

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

```
/* Defines via a typedef */
typedef char __data20 D20Byte;
typedef D20Byte *D20BytePtr;
D20Byte aByte;
D20BytePtr 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:

```
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.

```
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 are 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 301.

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 `data16` memory.

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

__data16 struct MyStruct gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
    int mAlpha;
    __data16 int mBeta; /* Incorrect declaration */
};
```

MORE EXAMPLES

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

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

<code>int myA;</code>	A variable stored in default memory determined by the data model in use.
<code>int __data16 myB;</code>	A variable stored in data16 memory.
<code>__data20 int myC;</code>	A variable stored in data20 memory.
<code>int * myD;</code>	A pointer stored in default memory. The pointer points to an integer in default memory.
<code>int __data16 * myE;</code>	A pointer stored in default memory. The pointer points to an integer in data16 memory.
<code>int __data16 * __data20 myF;</code>	A pointer stored in data20 memory pointing to an integer stored in data16 memory.
<code>int __data20 * MyFunction(int __data16 *);</code>	A declaration of a function that takes a parameter which is a pointer to an integer stored in data16 memory. The function returns a pointer to an integer stored in data20 memory.

C++ AND MEMORY TYPES

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

In non-static member functions, the non-static member variables of a C++ object can be referenced via the `this` pointer, explicitly or implicitly. The `this` pointer is of the default data pointer type unless class memory is used, see Using IAR attributes with Classes, page 203.

In the Medium data model, this means that unless class memory is used, 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 Using IAR attributes with Classes, page 203.

Data models

Data models are only available for the MSP430X architecture.

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

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

The data model specifies the default memory type. 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 for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 61.

Note: Your choice of data model does not affect the placement of code.

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 and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 61.

This table summarizes the different data models:

Data model name	Default memory attribute	Data20 available
Small	<code>__data16</code>	No
Medium	<code>__data16</code>	Yes
Large	<code>__data20</code>	Yes

Table 4: Data model characteristics



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



Use the `--data_model` option to specify the data model for your project; see `--data_model`, page 264.

The Small data model

The Small data model is designed to save stack space when you do not need to put any data outside the low 64 kBytes of memory. It uses the `data16` memory type only. The `data20` memory type is not allowed. This means that normal (non-extended) instructions are used for all memory accesses. However, the compiler can use extended instructions when they are more efficient, for other types of operations, like for example shifts.

The 20-bit registers in the MSP430X architecture are treated as if they were 16 bits wide, and the compiler assumes that the most significant 4 bits are never used. Only the least significant 16 bits are preserved across function calls when mandated by the calling convention. In practice, an interrupt handler could clear those upper 4 bits at any time.

Note: There are ways to make interrupt handlers in the Small data model preserve the full 20 bits of every register, see `--save_reg20`, page 289 and `__save_reg20`, page 321. However, because these are rather inefficient, you should consider selecting the Medium data model instead if you need the full 20 bits of registers to be preserved.

If the Small data model is used in combination with the Large code model, 20-bit function pointers are passed as pairs of two 16-bit registers in the calling convention.

To access data outside the low 64 kBytes of memory in the Small data model, you must use intrinsic functions.

The Medium data model

The Medium data model provides good efficiency when most but not all data is placed in the low 64 kBytes of memory, and has a natural way to access the data that is outside the low 64 kBytes. It uses the `data16` memory type by default, but you can use the `data20` memory type through explicit overrides on individual variables and pointers. This means that normal (non-extended) instructions are used for all memory accesses except those where you have specified that the data could be placed outside the low 64 kBytes of memory.

The compiler is aware that the most significant 4 bits of the registers can contain valid data, and preserves the full 20 bits across function calls when mandated by the calling convention. Because of this, the Medium data model often uses more stack space than the Small data model.

Function pointers are always passed in a single register in the calling convention, regardless of the selected code model.

The Large data model

The Large data model automatically allows data to be placed outside the low 64 kBytes of memory, but requires careful tuning to be really efficient. It uses the data20 memory type by default, but you can use the data16 memory type through explicit overrides on individual variables and pointers. This means that extended instructions will be used for most memory accesses, unless you have specified that the data will be placed in the low 64 kBytes.

The compiler is aware that the most significant 4 bits of the registers can contain valid data, and preserves the full 20 bits across function calls when mandated by the calling convention.

Function pointers are always passed in a single register in the calling convention, regardless of the selected code model.

Storage of auto variables and parameters

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

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

THE STACK

The stack can contain:

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

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

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

See also *Stack considerations*, page 217 and *Setting up stack memory*, page 105.

Advantages

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

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

Potential problems

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

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

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

Dynamic memory on the heap

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

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

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

The compiler supports heaps in more than one memory types:

Memory type	Memory attribute	Section name	Used by default in data model
data16	__data16	DATA16_HEAP	Small and Medium
data20	__data20	DATA20_HEAP	Large

Table 5: Heaps supported in memory types

In DLIB, to use a specific heap, add the memory attribute in front of `malloc`, `free`, `alloc`, and `realloc`, for example `__data16_malloc`. The default functions will use of the specific heap variants, depending on project settings such as data model.

For information about how to set up the size for heap memory, see *Setting up heap memory*, page 106 .

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

- Function-related extensions
- Code models (MSP430X only)
- Primitives for interrupts, concurrency, and OS-related programming
- Execution in RAM
- Position-independent code and read-only data
- Inlining functions

Function-related extensions

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

- Control call and return sequences
- Execute functions in RAM
- Use primitives for interrupts, concurrency, and OS-related programming
- Control function inlining
- Facilitate function optimization
- Access hardware features.

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

For more information about optimizations, see *Efficient coding for embedded applications*, page 227. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models (MSP430X only)

Use *code models* to specify how functions are called. Technically, the code models control the following:

- The instruction sequences used for calling and returning from functions
- The memory range where functions can be stored

- The size of function pointers.

Your project can only use one code model at a time, and the same model must be used by all user modules and all library modules.

Note: Your choice of code model does not affect the placement of data.

These code models are available:

Code model name	Instructions	Memory range	Function pointer size
Small	CALL, RET	64 Kbytes	2 bytes
Large (default)	CALLA, RETA	1 Mbyte	4 bytes

Table 6: Code models

If you do not specify a code model, the compiler will use the Large code model as default.



See the *IDE Project Management and Building Guide* for information about specifying a code model in the IDE.



Use the `--code_model` option to specify the code model for your project; see `--code_model`, page 262.

THE SMALL CODE MODEL

The Small code model uses standard 16-bit instructions for call and return. Functions must be placed in the first 64 Kbytes of memory.

The function pointers are 2 bytes in memory.

THE LARGE CODE MODEL

The Large code model uses extended 20-bit instructions for call and return. Functions can be placed anywhere in the 1 Mbyte of memory.

In the Small data model, a function pointer is treated as a 32-bit value. It occupies 4 bytes of memory and a pair of registers is required to hold it.

In the Medium and Large data models, a function pointer is a 20-bit value. In memory, it occupies 4 bytes but it fits into one 20-bit register.

COMPARING THE CODE MODELS

The main difference between the code models is that the Small code model is slightly faster, it requires less runtime stack, and pointers are smaller and require less memory. However, in the Small code model code placement is restricted to the first 64 Kbytes.

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`, `__save_reg20`
- The pragma directives: `#pragma vector`, `#pragma no_epilogue`
- The intrinsic functions: `__enable_interrupt`, `__disable_interrupt`.
- 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 routine, 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 can for example contain 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`.

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

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

The header files `iodevice.h` and `mspdevice.h`, where *device* corresponds to the selected device, contain predefined names for the existing interrupt vectors.

Defining an interrupt function—an example

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

```
#pragma vector = TIMERA0_VECTOR /* Defined in I/O header file */
__interrupt void MyInterruptRoutine(void)
{
    /* Do something */
}
```

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

Interrupt and C++ member functions

Only `static` member functions can be interrupt functions. 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.

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:

```
__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:

```
#pragma vector=TIMER_A1_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 applied to any other value, the effect is undefined and the application will most likely fail.

For more information about the intrinsic keyword, see `__even_in_range`, page 354.

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, interrupt routines preserve the low 16 bits of all registers. However, the upper 4 bits could be cleared. Therefore, 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 more information, see `__monitor`, page 315.



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.

```

/* 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();
    }
}

```

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 135.

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.

Position-independent code and read-only data

Most applications are designed to be placed at a fixed position in memory. The exact placement of each function and variable is decided at link time. However, sometimes it is useful to instead decide at runtime where to place the application, for example in certain systems where applications are loaded dynamically.

The compiler supports a feature known as *position-independent code and read-only data* (ROPI). When the `--ropi` option is specified, the compiler generates code that uses PC-relative references for addressing code and read-only data. This means that, even though the linker places the code and read-only data at fixed locations, the application can still be executed correctly when the linked image is placed at a different address than where it was linked.

Note: Only functions and read-only data are affected—variables in RAM are never position-independent. This usually means that only one ROPI application at a time can execute.

DRAWBACKS AND LIMITATIONS

There are some drawbacks and limitations to bear in mind when using ROPI:

- The code generated for function calls and accesses to read-only data will be somewhat larger
- Data initialization at startup might be somewhat slower, and the initialization data might be somewhat larger
- Read-only global or static pointer variables that point to read-only data are not supported
- C++ is not supported
- Interrupt handlers that use the `#pragma vector` directive are not handled automatically
- The CLIB runtime library cannot be used
- The object attributes `__ramfunc` and `__persistent` are not supported.

Note: In some cases, there is an alternative to ROPI that avoids these limitations and drawbacks: If there is only a small number of possible placements for the application, you can compile the application without `--ropi` and link it multiple times, once for each possible placement. Then you can choose the correct image at load time. When applicable, this approach makes better use of the available hardware resources than using ROPI.

INITIALIZATION OF POINTERS

In a ROPI application, the actual addresses of read-only data variables are only known at runtime. A pointer to data always holds the actual address, so that it can be used efficiently in indirect and indexed addressing modes. This means that pointers to read-only data cannot be initialized statically (by copying from ROM at startup). For non-constant global and static pointer variables, the compiler automatically rewrites static initializations to dynamic ones, where the actual address is computed at program startup. This rewriting can be disabled if you prefer the compiler to issue an error message at compile-time, see `--no_rw_dynamic_init`, page 279.

Unlike data pointers, function pointers in a ROPI application hold the linked address of a function, and the actual address is only computed when the function is called. This means that function pointers work as usual without limitations.

NON-ROPI CODE IN ROPI SYSTEMS

In a system with ROPI applications, there might be a small amount of non-ROPI code that handles startup, dynamic loading of applications, shared firmware functions, etc. Such code must be compiled and linked separately from the ROPI applications. You can use the function type attribute `__no_pic` to inform the compiler that a function pointer holds the actual address of a non-ROPI function. This allows a ROPI application to call non-ROPI functions via `__no_pic` function pointers. See `__no_pic`, page 318.

Inlining functions

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

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

C VERSUS C++ SEMANTICS

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

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

For example:

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

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

FEATURES CONTROLLING FUNCTION INLINING

There are several mechanisms for controlling function inlining:

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

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

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

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

By using the `#pragma inline` directive you can also disable the compiler's heuristics to either force inlining or completely disable inlining. For more information, see `inline`, page 334.

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

The compiler can only inline a function if the definition is known. Normally, this is restricted to the current translation unit. However, when the `--mfc` compiler option for multi-file compilation is used, the compiler can inline definitions from all translation units in the multi-file compilation unit. For more information, see [Multi-file compilation units](#), page 235.

For more information about the function inlining optimization, see [Function inlining](#), page 238.

Linking overview

- Linking—an overview
- Segments and memory
- The linking process in detail
- Placing code and data—the linker configuration file
- Initialization at system startup
- Stack usage analysis

Linking—an overview

The IAR XLINK Linker is a powerful, flexible software tool for use in the development of embedded applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C/C++, or mixed C/C++ and assembler programs.

The linker combines one or more relocatable object files—produced by the IAR Systems compiler or assembler—with required parts of object libraries to produce an executable image containing machine code for the microcontroller you are using. XLINK can generate more than 30 industry-standard loader formats, in addition to the proprietary format UBROF which is used by the C-SPY debugger.

The linker will automatically load only those library modules that are actually needed by the application you are linking. Further, the linker eliminates segment parts that are not required. During linking, the linker performs a full C-level type checking across all modules.

The linker uses a *configuration file* where you can specify separate locations for code and data areas of your target system memory map.

The final output produced by the linker is an absolute, target-executable object file that can be downloaded to the microcontroller, to C-SPY, or to a compatible hardware debugging probe. Optionally, the output file can contain debug information depending on the output format you choose.

To handle libraries, the library tools XAR and XLIB can be used.

Segments and memory

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

WHAT IS A SEGMENT?

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

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

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

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

Segment memory type

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

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

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

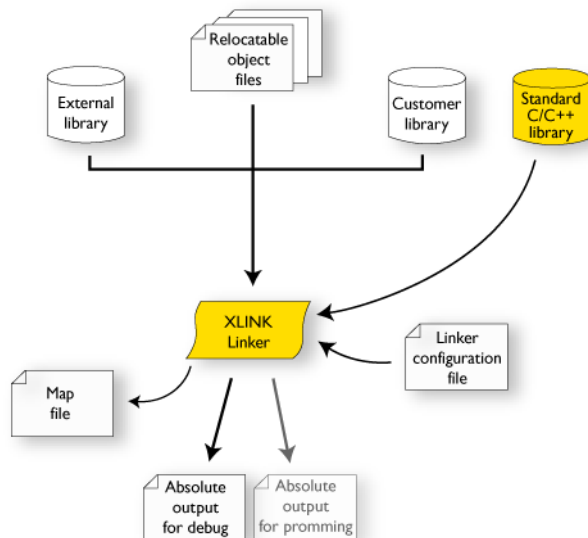
The linking process in detail

The relocatable modules in object files and libraries, produced by the IAR compiler and assembler, cannot be executed as is. To make an application executable, the object files must be *linked*.

The IAR XLINK Linker is used for the link process. It normally performs the following procedure (note that some of the steps can be turned off by command line options or by directives in the linker configuration file):

- Determines which modules to include in the application. Program modules are always included. Library modules are only included if they provide a definition for a global symbol that is referenced from an included module. If the object files containing library modules contain multiple definitions of variables or functions, only the first definition will be included. This means that the linking order of the object files is important.
- Determines which segment parts from the included modules to include in the application. Only those segments that are actually needed by the application are included. There are several ways to determine of which segment parts that are needed, for example, the `__root` object attribute, the `#pragma required` directive, and the `-g` linker option.
- Divides each segment that will be initialized by copying into two segments, one for the ROM part and one for the RAM part. The RAM part contains the label and the ROM part the actual bytes. The bytes are conceptually linked as residing in RAM.
- Determines where to place each segment according to the segment placement directives in the *linker configuration file*.
- Produces an absolute file that contains the executable image and any debug information. The contents of each needed segment in the relocatable input files is calculated using the relocation information supplied in its file and the addresses determined when placing segments. This process can result in one or more range errors if some of the requirements for a particular segment are not met, for instance if placement resulted in the destination address for a PC-relative jump instruction being out of range for that instruction.
- Optionally, produces a map file that lists the result of the segment placement, the address of each global symbol, and finally, a summary of memory usage for each module.

This illustration shows the linking process:



During the linking, XLINK might produce error messages and optionally a map file. In the map file you can see the result of the actual linking and is useful for understanding why an application was linked the way it was, for example, why a segment part was included. If a segment part is not included although you expect it to be, the reason is *always* that the segment part was not referenced to from an included part.

Note: To inspect the actual content of the object files, use XLIB. See the *IAR Linker and Library Tools Reference Guide*.

Placing code and data—the linker configuration file

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 microcontroller. To use the same source code with different devices, just rebuild the code with the appropriate linker configuration file.

In particular, the linker configuration file specifies:

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

- The maximum heap size.

The file consists of a sequence of linker commands. This means that the linking process will be governed by all commands in sequence.

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

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

Note: The supplied linker configuration file includes comments explaining the contents.

For more information about the linker configuration file and how to customize it, see *Linking considerations*, page 101.

See also the *IAR Linker and Library Tools Reference Guide*.

Initialization at system startup

In Standard C, all static variables—variables that are allocated at a fixed memory address—must be initialized by the runtime system to a known value at application startup. Static variables can be divided into these categories:

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero
- Variables that are located by use of the @ operator or the #pragma location directive
- Variables that are declared as const and therefore can be stored in ROM

- Variables defined with the `__no_init` keyword, meaning that they should not be initialized at all.

STATIC DATA MEMORY SEGMENTS

The compiler generates a specific type of segment for each type of variable initialization.

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`.

Some of the declared data is placed in non-volatile memory, for example ROM/flash, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more information about segment memory types, see *Segment memory type*, page 86.

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

Categories of declared data	Source	Segment type	Segment name*	Segment content
Zero-initialized data	<code>int i;</code>	Read/write data, zero-init	<code>MEMATTR_Z</code>	None
Zero-initialized data	<code>int i = 0;</code>	Read/write data, zero-init	<code>MEMATTR_Z</code>	None
Initialized data (non-zero)	<code>int i = 6;</code>	Read/write data	<code>MEMATTR_I</code>	None
Initialized data (non-zero)	<code>int i = 6;</code>	Read-only data	<code>MEMATTR_ID</code>	Initializer data for <code>MEMATTR_I</code>
Constants	<code>const int i = 6;</code>	Read-only data	<code>MEMATTR_C</code>	The constant
Non-initialized data	<code>__no_init int i;</code>	Read/write data	<code>MEMATTR_N</code>	None
Persistent data	<code>__persistent int i = 6;</code>	Read-only data†	<code>MEMATTR_P</code>	The constant
Non-initialized absolute addressed data	<code>__no_init int i @ 0x200;</code>	Read/write data	<code>MEMATTR_AN</code>	None

Table 7: segments holding initialized data

Categories of declared data	Source	Segment type	Segment name*	Segment content
Constant absolute addressed data	<code>const int i @ 0x200 = 10000;</code>	Read-only data	<code>MEMATTR_AC</code>	The constant

Table 7: segments holding initialized data (Continued)

* The actual segment group name—`MEMATTR`—depends on the memory where the variable is placed. See *Memory types (MSP430X only)*, page 60.

† Only read-only segments can contain data, even though the application can write to `--persistent` data.

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

THE INITIALIZATION PROCESS

Initialization of data is handled by the system startup code. If you add more segments, you must update the system startup code accordingly.

To configure the initialization of variables, you must consider these issues:

- Segments that should be zero-initialized should only be placed in RAM.
- Segments that should be initialized, except for zero-initialized segments:

The system startup code 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 very 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.
- Segments that contain constants do not need to be initialized; they should only be placed in flash/ROM
- Segments holding `__no_init` declared variables should not be initialized.
- Finally, global C++ object constructors are called.

For more information about and examples of how to configure the initialization, see *Linking considerations*, page 101.

Stack usage analysis

This section describes how to perform a stack usage analysis using the linker.

In the `MSP430\src` directory, you can find an example project that demonstrates stack usage analysis.

INTRODUCTION TO STACK USAGE ANALYSIS

Under the right circumstances, the linker can accurately calculate the maximum stack usage for each call graph, starting from the program start, interrupt functions, tasks etc. (each function that is not called from another function, in other words, the root).

If you enable stack usage analysis, a stack usage chapter will be added to the linker map file, listing for each call graph root the particular call chain which results in the maximum stack depth.

The analysis is only accurate if there is accurate stack usage information for each function in the application.

In general, the compiler will generate this information for each C function, but if there are indirect calls (calls using function pointers) in your application, you must supply a list of possible functions that can be called from each calling function.

If you use a stack usage control file, you can also supply stack usage information for functions in modules that do not have stack usage information.

You can use the `check that` directive in your stack usage control file to check that the stack usage calculated by the linker does not exceed the stack space you have allocated.

PERFORMING A STACK USAGE ANALYSIS

- 1 Enable stack usage analysis:



In the IDE, choose **Project>Options>Linker>Advanced>Enable stack usage analysis**



On the command line, use the linker option `--enable_stack_usage`

See the *IAR Linker and Library Tools Reference Guide* for information about the linker option `--enable_stack_usage`.

- 2 Enable the linker map file:



In the IDE, choose **Project>Options>Linker>List>Generate linker listing**



On the command line, use the linker option `-l`

- 3 Link your project. Note that the linker will issue warnings related to stack usage under certain circumstances, see *Situations where warnings are issued*, page 97.

- 4 Review the linker map file, which now contains a stack usage chapter with a summary of the stack usage for each call graph root. For more information, see *Result of an analysis—the map file contents*, page 93.
- 5 For more details, analyze the call graph log, see *Call graph log*, page 97.

Note that there are limitations and sources of inaccuracy in the analysis, see *Limitations*, page 96.

You might need to specify more information to the linker to get a more representative result. See *Specifying additional stack usage information*, page 95



In the IDE, choose **Project>Options>Linker>Advanced>Enable stack usage analysis>Control file**



On the command line, use the linker option `--stack_usage_control`

See the *IAR Linker and Library Tools Reference Guide* for information about the linker option `--stack_usage_control`.

- 6 To add an automatic check that you have allocated memory enough for the stack, use the `check that` directive in your stack usage control file. For example, assuming a stack segment named `MY_STACK`, you can write like this:

```
check that size("MY_STACK") >=maxstack("Program entry")
                                     + totalstack("interrupt") + 100;
```

When linking, the linker emits an error if the check fails. In this example, an error will be emitted if the sum of the following exceeds the size of the `MY_STACK` segment:

- The maximum stack usage in the category `Program entry` (the main program).
- The sum of each individual maximum stack usage in the category `interrupt` (assuming that all interrupt routines need space at the same time).
- A safety margin of 100 bytes (to account for stack usage not visible to the analysis).

See also `check that` directive, page 398 and *Stack considerations*, page 217.

RESULT OF AN ANALYSIS—THE MAP FILE CONTENTS

When stack usage analysis is enabled, the linker map file contains a stack usage chapter with a summary of the stack usage for each call graph root category, and lists the call

chain that results in the maximum stack depth for each call graph root. This is an example of what the stack usage chapter in the map file might look like:

```
*****
*
*                               STACK USAGE ANALYSIS
*
*                               *
*****

Call Graph Root Category  Max Use  Total Use
-----
interrupt                 4         4
Program entry             350       350

Program entry
  "__program_start": 0x0000c0f8

Maximum call chain                               350 bytes

  "__program_start"                               0
  "main"                                           4
  "WriteObject"                                   24
  "DumpObject"                                    0
  "PrintObject"                                   8
  "fprintf"                                       4
  "_PrintfLarge"                                 126
  "_PutstrLarge"                                 100
  "pad"                                           14
  "_PutcharsLarge"                               10
  "_FProut"                                       6
  "fputc"                                         6
  "_Fwprep"                                       6
  "fseek"                                         4
  "_Fspos"                                        14
  "fflush"                                        6
  "fflushOne"                                    6
  "__write"                                       0
  "__dwrite"                                     10
  "__DebugBreak"                                 2

interrupt
  "DoStuff()": 0x0000e9ee

Maximum call chain                               4 bytes

  "DoStuff()"                                     4
```

The summary contains the depth of the deepest call chain in each category as well as the sum of the depths of the deepest call chains in that category.

Each call graph root belongs to a call graph root category to enable convenient calculations in `check that` directives.

SPECIFYING ADDITIONAL STACK USAGE INFORMATION

To specify additional stack usage information you can use either a stack usage control file (`suc`) where you specify stack usage control directives or annotate the source code.

You can:

- Specify complete stack usage information (call graph root category, stack usage, and possible calls) for a function, by using the stack usage control directive `function`. Typically, you do this if stack usage information is missing, for example in an assembler module. In your `suc` file you can for example write like this:

```
function MyFunc: 32,
    calls MyFunc2,
    calls MyFunc3, MyFunc4: 16;
```

```
function [interrupt] MyInterruptHandler: 44;
```

See also `function` directive, page 399.

- Exclude certain functions from stack usage analysis, by using the stack usage control directive `exclude`. In your `suc` file you can for example write like this:

```
exclude MyFunc5, MyFunc6;
```

See also `exclude` directive, page 398.

- Specify a list of possible destinations for indirect calls in a function, by using the stack usage control directive `possible calls`. Use this for functions which are known to perform indirect calls and where you know exactly which functions that might be called in this particular application. In your `suc` file you can for example write like this:

```
possible calls MyFunc7: MyFunc8, MyFunc9;
```

If the information about which functions that might be called is available at compile time, consider using the `#pragma calls` directive instead.

See also `possible calls` directive, page 400 and `calls`, page 327.

- Specify that functions are call graph roots, including an optional call graph root category, by using the stack usage control directive `call graph root` or the `#pragma call_graph_root` directive. In your `suc` file you can for example write like this:

```
call graph root [task]: MyFunc10, MyFunc11;
```

If your interrupt functions have not already been designated as call graph roots by the compiler, you must do so manually. You can do this either by using the `#pragma call_graph_root` directive in your source code or by specifying a directive in your `suc` file, for example:

```
call graph root [interrupt]: Irq1Handler, Irq2Handler;
```

See also `call graph root` directive, page 397 and `call_graph_root`, page 327.

- Specify a maximum number of iterations through any of the cycles in the recursion nest of which the function is a member. In your `suc` file you can for example write like this:

```
max recursion depth MyFunc12: 10;
```

- Selectively suppress the warning about unmentioned functions referenced by a module for which you have supplied stack usage information in the stack usage control file. Use the `no calls from` directive in your `suc` file, for example like this:
- ```
no calls from [file.r43] to MyFunc13, MyFunc14;
```
- Instead of specifying stack usage information about assembler modules in a stack usage control file, you can annotate the assembler source with call frame information. For more information, see the *IAR Assembler User Guide for MSP430*.

For more information, see the chapter `The stack usage control file`, page 397.

## LIMITATIONS

Apart from missing or incorrect stack usage information, there are also other sources of inaccuracy in the analysis:

- The linker cannot always identify all functions in object modules that lack stack usage information. In particular, this might be a problem with object modules written in assembly language. You can provide stack usage information for such modules using a stack usage control file, and for assembly language modules you can also annotate the assembler source code with `CFI` directives to provide stack usage information. See the *IAR Assembler User Guide for MSP430*.
- If you use inline assembler to change the frame size or to perform function calls, this will not be reflected in the analysis.
- Extra space consumed by other sources (the processor, an operating system, etc) is not accounted for.
- The support for C++ is very incomplete. In particular virtual function calls, dynamic objects, and various indirect calls from the standard library (like the functions set up by `set_new_handler` and `register_callback`) are not supported at all.
- If you use other forms of function calls, they will not be reflected in the call graph.



- Using multi-file compilation (`--mfc`) can interfere with using a stack usage control file to specify properties of module-local functions in the involved files.

Note that stack usage analysis produces a worst case result. The program might not actually ever end up in the maximum call chain, by design, or by coincidence.



Stack usage analysis is only a complement to actual measurement. If the result is important, you need to perform independent validation of the results of the analysis.

## SITUATIONS WHERE WARNINGS ARE ISSUED

When stack usage analysis is enabled in the linker, warnings will be generated in the following circumstances:

- There is a function without stack usage information.
- There is an indirect call site in the application for which a list of possible called functions has not been supplied.
- There are no known indirect calls, but there is an uncalled function that is not known to be a call graph root.
- The application contains recursion (a cycle in the call graph) for which no maximum recursion depth has been supplied, or which is of a form for which the linker is unable to calculate a reliable estimate of stack usage.
- There are calls to a function declared as a call graph root.
- You have used the stack usage control file to supply stack usage information for functions in a module that does not have such information, and there are functions referenced by that module which have not been mentioned as being called in the stack usage control file.

## CALL GRAPH LOG

To help you interpret the results of the stack usage analysis, there is a log output option that produces a simple text representation of the call graph (`--log stack_usage`).

## Example output:

```

Program entry:
0 __program_start [350]
 0 __data16_memzero [2]
 2 - [0]
 0 __data16_memcpy [2]
 0 memcpy [2]
 2 - [0]
 2 - [0]
 0 main [350]
 4 ParseObject [52]
 28 GetObject [28]
 34 getc [22]
 38 _Frprep [18]
 44 malloc [12]
 44 __data16_malloc [12]
 48 __data16_findmem [8]
 52 __data16_free [4]
 56 - [0]
 52 __data16GetMemChunk [2]
 54 - [0]
 46 - [0]
 44 __read [12]
 54 __DebugBreak [2]
 56 - [0]
 36 - [0]
 34 CreateObject [18]
 40 malloc [12] ***
 4 ProcessObject [326]
 8 ProcessHigh [76]
 34 ProcesMedium [50]
 60 ProcessLow [24]
 84 - [0]
 8 DumpObject [322]
 8 PrintObject [322]
 16 fprintf [314]
 20 _PrintfLarge [310]
 10 - [0]
 4 WriteObject [346]
 28 DumpObject [322] ***
 4 DestroyObject [28]
 28 free [4]
 28 __data16_free [4] ***
 30 - [0]
 0 exit [38]
 0 _exit [38]
 4 _Close_all [34]

```

```

8 fclose [30]
 14 _Fofree [4]
 14 free [4] ***
 16 - [0]
 14 fflush [24] ***
 14 free [4] ***
 14 __close [8]
 20 __DebugBreak [2] ***
 14 remove [8]
 20 __DebugBreak [2] ***
8 __write [12] ***
2 __exit [8]
8 __DebugBreak [2] ***
2 - [0]

```

Each line consists of this information:

- The stack usage at the point of call of the function
- The name of the function, or a single '-' to indicate usage in a function at a point with no function call (typically in a leaf function)
- The stack usage along the deepest call chain from that point. If no such value could be calculated, "[---]" is output instead. "\*\*\*" marks functions that have already been shown.



# Linking your application

- Linking considerations
- Checking module consistency

---

## Linking considerations



When you set up your project in the IAR Embedded Workbench IDE, a default linker configuration file is automatically used based on your project settings and you can simply link your application. For the majority of all projects it is sufficient to configure the vital parameters that you find in **Project>Options>Linker>Config**.



When you build from the command line, you can use a ready-made linker command file provided with your product package.

The `config` directory contains ready-made linker configuration files for all supported devices (filename extension `.xcl`). The files contain the information required by XLINK, and are ready to be used as is. The only change, if any, you will normally have to make to the supplied 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 also add details about the external RAM memory area.

To edit a linker configuration file, use the editor in the IDE, or any other suitable editor. Do not change the original template file. We recommend that you make a copy in the working directory, and modify the copy instead.

If you find that the default linker configuration file does not meet your requirements, you might want to consider:

- Placing segments
- Placing data
- Setting up stack memory
- Setting up heap memory
- Placing code
- Keeping modules
- Keeping symbols and segments
- Application startup
- Interaction between XLINK and your application
- Producing other output formats than UBROF

## PLACING SEGMENTS

The placement of segments in memory is performed by the IAR XLINK Linker.

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

In demonstrating the methods, fictitious examples are used based on this memory layout:

- There is 1 Mbyte addressable memory.
- There is ROM memory in the address ranges 0x0000–0x1FFF, 0x3000–0x4FFF, and 0x10000–0x1FFFF.
- There is RAM memory in the address ranges 0x8000–0xAFFF, 0xD000–0xFFFF, and 0x20000–0x27FFF.
- There are two addressing modes for data, one for data16 memory and one for data20 memory.
- There is one stack and one heap.
- There are two addressing modes for code, one for near\_func memory and one for far\_func memory.

**Note:** Even though you have a different memory map, for example if you have additional memory spaces (EEPROM) and additional segments, you can still use the methods described in the following examples.

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.

For the result of each placement directive after linking, inspect the segment map in the list file (created by using the command line option `-x`).

### General hints for placing segments

When you consider where in memory you should place your segments, it is typically a good idea to start placing large segments first, then placing small segments.

In addition, you should consider these aspects:

- Start placing the segments that must be placed on a specific address. This is, for example, often the case with the segment holding the reset vector.
- Then consider placing segments that hold content that requires continuous memory addresses, for example the segments for the stack and heap.

- When placing code and data segments for different addressing modes, make sure to place the segments in size order (the smallest memory type first).

**Note:** Before the linker places any segments in memory, the linker will first place the absolute segments.

### 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 `0x0000-0x1FFF`.

```
-Z (CONST) MYSEGMENTA, MYSEGMENTB=0000-1FFF
```

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

```
-Z (CONST) MYSEGMENTA=0000-1FFF
-Z (CODE) MYCODE
```

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

```
-Z (CONST) MYSMALLSEGMENT=0000-01FF
-Z (CONST) MYLARGESEGMENT=0000-1FFF
```



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=8000-AFFF, D000-FFFF
```

If your application has an additional RAM area in the memory range 0x6000-0x67FF, you can simply add that to the original definition:

```
-P (DATA) MYDATA=8000-AFFF, D000-FFFF, 6000-67FF
```

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

**Note:** Copy initialization segments—`BASENAME_I` and `BASENAME_ID`—and dynamic initialization segments must be placed using `-z`. For code placed in RAM, see *Code in RAM*, page 108.

## PLACING DATA

Static memory is memory that contains variables that are global or declared static.

### Placing static memory data segments

Depending on their memory attribute, static data is placed in specific segments. For information about the segments used by the compiler, see *Static data memory segments*, page 90.

For example, these commands can be used to place the static data segments:

```
/* First, the segments to be placed in ROM are defined. */
-Z (CONST) DATA16_C=0000-1FFF, 3000-4FFF
-Z (CONST) DATA20_C=0000-1FFF, 3000-4FFF, 10000-1FFFF
-Z (CONST) DATA16_ID, DATA20_ID=010000-1FFFF

/* Then, the RAM data segments are placed in memory. */
-Z (DATA) DATA16_I, DATA16_Z, DATA16_N=8000-AFFF
-Z (DATA) DATA20_I, DATA20_Z, DATA20_N=20000-27FFF
```

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

### Placing 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 for example either the `DATA16_AC` or the `DATA16_AN` segment. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker configuration file.



## Placing user-defined segments

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

## SETTING UP STACK MEMORY

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

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

For more information about stack memory, see *Stack considerations*, page 217.



### 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 dedicated text box.



### Stack size allocation from the command line

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

The linker configuration file sets up a constant, representing the size of the stack, at the beginning of the file. Specify the appropriate size for your application, in this example 512 bytes:

```
-D_CSTACK_SIZE=200 /* 512 bytes of stack size */
```

Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

In many linker configuration files provided with IAR Embedded Workbench, this line is prefixed with the comment character `//` because the IDE controls the stack size allocation. To make the directive take effect, remove the comment character.



### Placing the stack segment

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

```
-Z (DATA) CSTACK+_CSTACK_SIZE#8000-AFFF
```

#### 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. Typically, 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.

## SETTING UP HEAP MEMORY

The heap contains dynamic data allocated by the C function `malloc` (or a corresponding function) or the C++ operator `new`.

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

- The linker segments used for the heap, see `DATA16_HEAP`, page 385 and `DATA20_HEAP`, page 387
- The steps for allocating the heap size, which differs depending on which build interface you are using
- The steps involved for placing the heap segments in memory.

See also *Heap considerations*, page 217.

In this example, the data segment for holding the heap is called `HEAP`.



### 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 dedicated text box.



### Heap size allocation from the command line

The size of the `HEAP` segment is defined in the linker configuration file.

The linker configuration file sets up a constant, representing the size of the heap, at the beginning of the file. Specify the appropriate size for your application, in this example 1024 bytes:

```
-D_HEAP_SIZE=400 /* 1024 bytes for heap memory */
```

Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

In many linker configuration files provided with IAR Embedded Workbench, these lines are prefixed with the comment character `//` because the IDE controls the heap size allocation. To make the directive take effect, remove the comment character.

If you use a heap, you should allocate at least 512 bytes for it, to make it work properly.



### Placing the heap segment

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

```
-Z (DATA)HEAP+_HEAP_SIZE=8000-AFFF
```

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

## PLACING CODE

This section contains descriptions of the segments used for storing code and the interrupt vector table. For information about all segments, see *Summary of segments*, page 379.

### Startup code

In this example, the segment `CSTART` contains code used during system startup and termination, see *System startup and termination*, page 135. The segment parts must also be placed into one continuous memory space, which means that the `-P` segment directive cannot be used.

This line will place the `CSTART` segment at the address `0x1100`:

```
-Z (CODE) CSTART=1100
```

### Code for MSP430

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

```
/* MSP430 devices */
-Z (CODE) CODE=0000-1FDF,3000-4FFF
```

### Code for MSP430X

In the Small code model, normal code is placed in the `CODE16` segment. In the Large code model, `CODE` is used. Interrupt functions are placed in `ISR_CODE`. This too is a simple operation in the linker configuration file:

```
-Z (CODE) CODE16,ISR_CODE=0000-1FDF,3000-4FFF
-Z (CODE) CODE=0000-1FDF,3000-4FFF,10000-FFFFF
```

### Interrupt vectors

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

```
-Z (CONST) INTVEC=FFE0-FFFF
```

The system startup code places the reset vector in the `RESET` segment; the `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:

```
-Z (CONST) RESET=FFFE-FFFF
```

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

For more information about the interrupt vectors, see *Interrupt vectors and the interrupt vector table*, page 73.

### Code in RAM

The segment `CODE_I` holds code which executes in RAM and which was initialized by `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.

```
-QCODE_I=CODE_ID

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

### C++ dynamic initialization

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

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

For example:

```
-Z (CONST) DIFUNCT=0000-1FFF,3000-4FFF
```

`DIFUNCT` must be placed using `-z`. For more information, see *DIFUNCT*, page 390.

### KEEPING MODULES

If a module is linked as a program module, it is always kept. That is, it will contribute to the linked application. However, if a module is linked as a library module, it is

included only if it is symbolically referred to from other parts of the application that have been included. This is true, even if the library module contains a root symbol. To assure that such a library module is always included, use `-A` to make all modules in the file be treated as if they were program modules:

```
-A file.r43
```

Use `-C` to makes all modules in the file be treated as if they were library modules:

```
-C file.r43
```

## KEEPING SYMBOLS AND SEGMENTS

By default, XLINK removes any segments, segment parts, and global symbols that are not needed by the application. To retain a symbol that does not appear to be needed—or actually, the segment part it is defined in—you can either use the `__root` attribute on the symbol in your C/C++ source code or `ROOT` in your assembler source code, or use the XLINK option `-g`.

For information about included and excluded symbols and segment parts, inspect the map file (created by using the XLINK option `-xm`).

For more information about the linking procedure for keeping symbols and sections, see *The linking process in detail*, page 87.

## APPLICATION STARTUP

By default, the point where the application starts execution is defined by the `__program_start` label, which is defined to point at the start of code. The label is also communicated via the debugger information to any debugger.

To change the start point of the application to another label, use the XLINK option `-s`.

## INTERACTION BETWEEN XLINK AND YOUR APPLICATION

Use the XLINK option `-D` to define symbols that can be used for controlling your application. You can also use symbols to represent the start and end of a continuous memory area that is defined in the linker configuration file.

To change a reference to one symbol to another symbol, use the XLINK command line option `-e`. This is useful, for example, to redirect a reference from a non-implemented function to a stub function, or to choose one of several different implementations of a certain function.

For information about the addresses and sizes of all global (statically linked) symbols, inspect the entry list in the map file (the XLINK option `-xm`).

## PRODUCING OTHER OUTPUT FORMATS THAN UBROF

XLINK can generate more than 30 industry-standard loader formats, in addition to the proprietary format UBROF which is used by the C-SPY debugger. For a complete list, see the *IAR Linker and Library Tools Reference Guide*. To specify a different output format than the default, use the XLINK option `-F`. For example:

```
-F intel-standard
```

Note that it can be useful to use the XLINK `-O` option to produce two output files, one for debugging and one for burning to ROM/flash.

Note also that if you choose to enable debug support using the `-x` option for certain low-level I/O functionality for mechanisms like file operations on the host computer etc, such debug support might have an impact on the performance and responsiveness of your application. In this case, the debug build will differ from your release build due to the debug modules included.

---

## 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

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

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

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

### LINKER MAP FILE

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

- A segment map which lists all segments in address order

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

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

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

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

---

## Checking module consistency

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

For example, in the compiler, it is possible to specify the 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.

### RUNTIME MODEL ATTRIBUTES

A runtime attribute is a pair constituted of a named key and its corresponding value. Two modules can only be linked together if they have the same value for each key that they both define.

There is one exception: if the value of an attribute is `*`, then that attribute matches any value. The reason for this is that you can specify this in a module to show that you have considered a consistency property, and this ensures that the module does not rely on that property.

## Example

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

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

*Table 8: Example of runtime model attributes*

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

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

## USING RUNTIME MODEL ATTRIBUTES

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

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

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

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

Note that key names that start with two underscores are reserved by the compiler. For more information about the syntax, see `rtmodel`, page 341 and the *IAR Assembler User Guide for MSP430*, respectively.

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



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

## PREDEFINED RUNTIME ATTRIBUTES

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

| Runtime model attribute   | Value | Description                                                                                                                                                          |
|---------------------------|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__rt_version</code> |       | 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. |

*Table 9: Predefined runtime model attributes*

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

If you are using assembler routines in the C or C++ code, see the chapter *Assembler directives* in the *IAR Assembler User Guide for MSP430*.



# The DLIB runtime environment

- Introduction to the runtime environment
- Setting up the runtime environment
- Additional information on the runtime environment
- Managing a multithreaded environment

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 about CLIB, see the chapter *The CLIB runtime environment*.

---

## Introduction to the runtime environment

A *runtime environment* is the environment in which your application executes.

This section contains information about:

- *Runtime environment functionality*, page 115
- *Briefly about input and output (I/O)*, page 116
- *Briefly about C-SPY emulated I/O*, page 118
- *Briefly about retargeting*, page 118

### RUNTIME ENVIRONMENT FUNCTIONALITY

The *DLIB runtime environment* supports Standard C and C++ and consists of:

- The *C/C++ standard library*, both its interface (provided in the system header files) and its implementation.
- Startup and exit code.
- Low-level I/O interface for managing input and output (I/O).
- Special compiler support, for instance functions for switch handling or integer arithmetics.

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
  - Peripheral unit registers and interrupt definitions in include files
  - The MSP430 hardware multiplier peripheral unit
  - Memory protection unit (MPU) and intellectual property encapsulation (IPE).

Runtime environment functions are provided in one or more *runtime libraries*.

The runtime library is delivered both as and (depending on your product package) as source files. The prebuilt libraries are available in different *configurations* to meet various needs, see *Runtime library configurations*, page 126. You can find the libraries in the product subdirectories `430\lib` and `430\src\lib`, respectively.

For more information about the library, see the chapter *C/C++ standard library functions*.

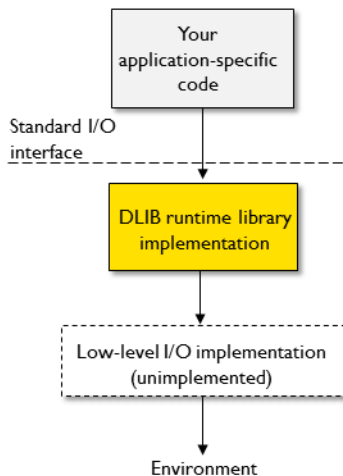
## BRIEFLY ABOUT INPUT AND OUTPUT (I/O)

Every application must communicate with its environment. The application might for example display information on an LCD, read a value from a sensor, get the current date from the operating system, etc. Typically, your application performs I/O via the C/C++ standard library or some third-party library.

There are many functions in the C/C++ standard library that deal with I/O, including functions for: standard character streams, file system access, time and date, miscellaneous system actions, and termination and assert. This set of functions is referred to as the *standard I/O interface*.

On a desktop computer or a server, the operating system is expected to provide I/O functionality to the application via the standard I/O interface in the runtime environment. However, in an embedded system, the runtime library cannot assume that such functionality is present, or even that there is an operating system at all. Therefore,

the low-level part of the standard I/O interface is not completely implemented by default:



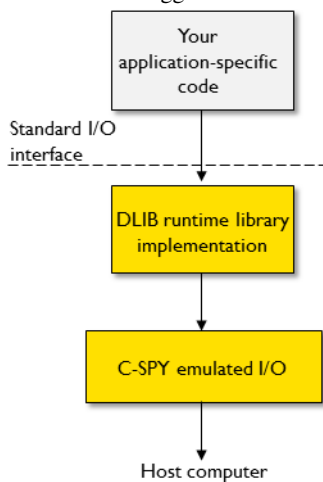
To make the standard I/O interface work, you can:

- Let the C-SPY debugger emulate I/O operations on the host computer, see *Briefly about C-SPY emulated I/O*, page 118
- *Retarget* the standard I/O interface to your target system by providing a suitable implementation of the interface, see *Briefly about retargeting*, page 118.

It is possible to mix these two approaches. You can, for example, let debug printouts and asserts be emulated by the C-SPY debugger, but implement your own file system. The debug printouts and asserts are useful during debugging, but no longer needed when running the application stand-alone (not connected to the C-SPY debugger).

## BRIEFLY ABOUT C-SPY EMULATED I/O

*C-SPY emulated I/O* is a mechanism which lets the runtime environment interact with the C-SPY debugger to emulate I/O actions on the host computer:



For example, when C-SPY emulated I/O is enabled:

- Standard character streams are directed to the C-SPY **Terminal I/O** window
- File system operations are performed on the host computer
- Time and date functions return the time and date of the host computer
- Termination and failed asserts break execution and notify the C-SPY debugger.

This behavior 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.

See *Setting up your runtime environment*, page 120 and *The C-SPY emulated I/O mechanism*, page 131.

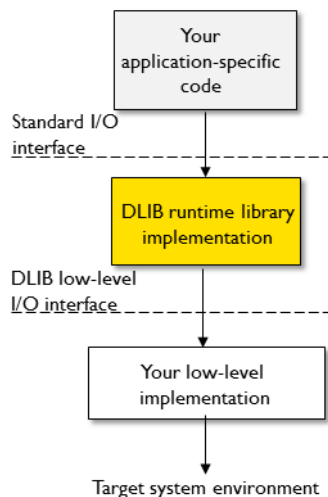
## BRIEFLY ABOUT RETARGETING

*Retargeting* is the process where you adapt the runtime environment so that your application can execute I/O operations on your target system.

The standard I/O interface is large and complex. To make retargeting easier, the DLIB runtime environment is designed so that it performs all I/O operations through a small set of simple functions, which is referred to as the *DLIB low-level I/O interface*. By default, the functions in the low-level interface lack usable implementations. Some are

unimplemented, others have stub implementations that do not perform anything except returning error codes.

To retarget the standard I/O interface, all you have to do is to provide implementations for the functions in the DLIB low-level I/O interface.



For example, if your application calls the functions `printf` and `fputc` in the standard I/O interface, the implementations of those functions both call the low-level function `__write` to output individual characters. To make them work, you just need to provide an implementation of the `__write` function—either by implementing it yourself, or by using a third-party implementation.

For information about how to override library modules with your own implementations, see *Overriding library modules*, page 123. See also *The DLIB low-level I/O interface*, page 139 for information about the functions that are part of the interface.

---

## Setting up the runtime environment

This section contains these tasks:

- *Setting up your runtime environment*, page 120
  - A runtime environment with basic project settings to be used during the initial phase of development.
- *Retargeting—Adapting for your target system*, page 122
- *Overriding library modules*, page 123

- *Customizing and building your own runtime library*, page 124

See also:

- *Managing a multithreaded environment*, page 152 for information about how to adapt the runtime environment to treat all library objects according to whether they are global or local to a thread.

## SETTING UP YOUR RUNTIME ENVIRONMENT

You can set up the runtime environment based on some basic project settings. It is also often convenient to let the C-SPY debugger manage things like standard streams, file I/O, and various other system interactions. This basic runtime environment can be used for simulation before you have any target hardware.

### To set up the runtime environment:

- 1 Before you build your project, choose **Project>Options>General Options** to open the **Options** dialog box.
- 2 On the **Library Configuration** page, verify the following settings:
  - **Library**: choose which *library* and *library configuration* to use. Typically, choose **Normal DLIB** or **Full DLIB**.  
For information about the various library configurations, see *Runtime library configurations*, page 126.
- 3 On the **Library Options** page, select **Auto** for both **Printf formatter** and **Scanf formatter**. This means that the linker will automatically choose the appropriate formatters based on information from the compiler. For more information about the available formatters and how to choose one manually, see *Formatters for printf*, page 129 and *Formatters for scanf*, page 130, respectively.
- 4 To enable C-SPY emulated I/O, choose **Project>Options>Linker>Output** and select the **Format** option that matches the level of support you need. See *Briefly about C-SPY emulated I/O*, page 118. Choose between:

| Linker option in the IDE            | Linker command line option | Description                                                                                                                    |
|-------------------------------------|----------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| <b>Debug information for C-SPY</b>  | <code>-Fubrof</code>       | Debug support for C-SPY, but without any support for C-SPY emulated I/O.                                                       |
| <b>With runtime control modules</b> | <code>-r</code>            | The same as <code>-Fubrof</code> but also limited support for C-SPY emulated I/O handling program abort, exit, and assertions. |

Table 10: Debug information and levels of C-SPY emulated I/O



| Linker option in the IDE          | Linker command line option | Description                                                                                                                                                                  |
|-----------------------------------|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>With I/O emulation modules</b> | <code>-rt</code>           | Full support for C-SPY emulated I/O, which means the same as <code>-r</code> , but also support for I/O handling, and accessing files on the host computer during debugging. |

Table 10: Debug information and levels of C-SPY emulated I/O (Continued)

**Note:** The C-SPY **Terminal I/O** window is not opened automatically; you must open it manually. For more information about this window, see the *C-SPY® Debugging Guide for MSP430*.

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

For this reason, a replacement for the `__write` function called `__write_buffered` is included in the runtime 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 in the IDE, choose **Project>Options>Linker>Output** and select the option **Buffered terminal output**.



To enable this function on the command line, add this to the linker command line:

```
-e__write_buffered=__write
```

- 6** Some math functions are available in different versions: default versions, smaller than the default versions, and larger but more accurate than default versions. Consider which versions you should use.

To specify which set of the math functions to use, choose **Project>Options>General Options>Library Options>Math functions** and choose which set to use. You can also specify individual functions.

For more information, see *Math functions*, page 131.

- 7** When you build your project, a suitable prebuilt library and library configuration file are automatically used based on the project settings you made.

For information about which project settings affect the choice of library file, see *Runtime library configurations*, page 126.

You have now set up a runtime environment that can be used while developing your application source code.

## RETARGETING—ADAPTING FOR YOUR TARGET SYSTEM

Before you can run your application on your target system, you must adapt some parts of the runtime environment, typically the system initialization and the DLIB low-level I/O interface functions.

### To adapt your runtime environment for your target system:

#### 1 Adapt system initialization.

It is likely that you must adapt the system initialization, for example, your application might need to initialize interrupt handling, I/O handling, watchdog timers, etc. You do this by implementing the routine `__low_level_init`, which is executed before the data segments are initialized. See *System startup and termination*, page 135 and *System initialization*, page 138. Note that you can find device-specific examples on this in the example projects provided in the product installation; see the Information Center.

#### 2 Adapt the runtime library for your target system. To implement such functions, you need a good understanding of the DLIB low-level I/O interface, see *Briefly about retargeting*, page 118.

Typically, you must implement your own functions if your application uses:

- Standard streams for input and output

If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you must implement your versions of the low-level functions `__read` and `__write`.

The low-level functions identify I/O streams, such as an open file, with a file handle that is a unique integer. The I/O streams normally associated with `stdin`, `stdout`, and `stderr` have the file handles 0, 1, and 2, respectively. When the handle is -1, all streams should be flushed. Streams are defined in `stdio.h`.

- File input and output

The library contains a large number of powerful functions for file I/O operations, such as `fopen`, `fclose`, `fprintf`, `fputs`, etc. All these functions call a small set of low-level functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters. Implement your version of these low-level functions.

- `signal` and `raise`

If the default implementation of these functions does not provide the functionality you need, you can implement your own versions.

- Time and date

To make the time and date functions work, you must implement the functions `clock`, `__time32`, `__time64`, and `__getzone`. Whether you use `__time32` or `__time64` depends on which interface you use for `time_t`, see `time.h`, page 376.

- Assert, see *\_ReportAssert*, page 144.
- Environment interaction
 

If the default implementation of `system` or `getenv` does not provide the functionality you need, you can implement your own versions.

For more information about the functions, see *The DLIB low-level I/O interface*, page 139.

The library files that you can override with your own versions are located in the `430\src\lib` directory.

- 3 When you have implemented your functions of the low-level I/O interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules*, page 123.

**Note:** If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 118.

- 4 Before you can execute your application on your target system, you must rebuild your project with a Release build configuration. This means that the linker will not include the C-SPY emulated I/O mechanism and the low-level I/O functions it provides. If your application calls any of the low-level functions of the standard I/O interface, either directly or indirectly, and your project does not contain these, the linker will issue an error for every missing low-level function. Also, note that the `NDEBUG` symbol is defined in a Release build configuration, which means asserts will no longer be generated. For more information, see *\_ReportAssert*, page 144.

## OVERRIDING LIBRARY MODULES

### To override a library function and replace it with your own implementation:

- 1 Use a template source file—a library source file or another template—and place a copy of it in your project directory.

The library files that you can override with your own versions are located in the `430\src\lib` directory.

- 2 Modify the file.

**Note:** To override the functions in a module, you must provide alternative implementations for all the needed symbols in the overridden module. Otherwise you will get error messages about duplicate definitions.

- 3 Add the modified file to your project, like any other source file.

**Note:** If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 118.

You have now finished the process of overriding the library module with your version.

## CUSTOMIZING AND BUILDING YOUR OWN RUNTIME LIBRARY

If the prebuilt library configurations do not meet your requirements, you can customize your own library configuration, but that requires that you *rebuild* the library.

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

- There is no prebuilt library for the required combination of compiler options or hardware support, for example, locked registers
- You want to build a library with direct support for the hardware multiplier
- You want to define your own library configuration with support for locale, file descriptors, multibyte characters, etc. This will include or exclude certain parts of the DLIB runtime environment.

In those cases, you must:

- Make sure that you have installed the library source code (`scr\lib`). If not already installed, you can install it using the IAR License Manager, see the *Installation and Licensing Guide*.
- Set up a library project
- Make the required library customizations
- Build your customized runtime library
- Finally, make sure your application project will use the customized runtime library.

### To set up a library project:

- I In the IDE, choose **Project>Create New Project** and use any of the library project templates that are available for the prebuilt libraries and that matches the project settings you need as closely as possible. See *Prebuilt runtime libraries*, page 127.

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.

- 2 Modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 54.

### To customize the library functionality:

- 1 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` which you can find in `430\inc\c`. This read-only file describes the configuration possibilities. Note that you should not modify this file.

In addition, your custom library has its own *library configuration file* `libraryname.h`—which you can find in `430\config\template\project`—and which sets up that specific library with the required library configuration. Customize this file by setting the values of the configuration symbols according to the application requirements. For example, to enable the `ll` qualifier in `printf` format strings, write in your library configuration file:

```
#define _DLIB_PRINTF_LONG_LONG 1
```

For information about configuration symbols that you might want to customize, see:

- *Configuration symbols for printf and scanf*, page 146
- *Configuration symbols for file input and output*, page 147
- *Locale*, page 147
- *Strtod*, page 150
- *Managing a multithreaded environment*, page 152

- 2 When you are finished, build your library project with the appropriate project options. After you build your library, you must make sure to use it in your application project.



To build IAR Embedded Workbench projects from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`). However, no make or batch files for building the library from the command line are provided. For information about the build process and the IAR Command Line Build Utility, see the *IDE Project Management and Building Guide*.

### To use the customized runtime library in your application project:

- 1 In the IDE, choose **Project>Options>General Options** and click the **Library Configuration** tab.
- 2 From the **Library** drop-down menu, choose **Custom DLIB**.
- 3 In the **Library file** text box, locate your library file.

- 4 In the **Configuration file** text box, locate your library configuration file.

---

## Additional information on the runtime environment

This section gives additional information on the runtime environment:

- *Runtime library configurations*, page 126
- *Prebuilt runtime libraries*, page 127
- *Formatters for printf*, page 129
- *Formatters for scanf*, page 130
- *The C-SPY emulated I/O mechanism*, page 131
- *Math functions*, page 131
- *The Texas Instruments MathLib library*, page 134
- *System startup and termination*, page 135
- *System initialization*, page 138
- *The DLIB low-level I/O interface*, page 139
- *Configuration symbols for printf and scanf*, page 146
- *Configuration symbols for file input and output*, page 147
- *Locale*, page 147
- *Strtod*, page 150
- *Hardware multiplier support*, page 150
- *MPU/IPE support*, page 150

### RUNTIME LIBRARY CONFIGURATIONS

The runtime library is provided with different *library configurations*, where each configuration is suitable for different application requirements.

The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The less functionality you need in the runtime environment, the smaller the environment becomes.

These predefined library configurations are available:

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

Table 11: Library configurations

**Note:** In addition to these predefined library configurations, you can provide your own configuration, see *Customizing and building your own runtime library*, page 124

If you do not specify a library configuration explicitly you will get the default configuration. If you use a prebuilt runtime library, a configuration file that matches the runtime library object file will automatically be used. See *Setting up your runtime environment*, page 120.

### To override the default library configuration, use one of these methods:

- 1 Use a prebuilt configuration of your choice—to specify a runtime configuration explicitly:



Choose **Project>Options>General Options>Library Configuration>Library** and change the default setting.



Use the `--dlib_config` compiler option, see `--dlib_config`, page 269.

The prebuilt libraries are based on the default configurations, see *Runtime library configurations*, page 126.

- 2 If you have built your own customized library, choose **Project>Options>Library Configuration>Library** and choose **Custom** to use your own configuration. For more information, see *Customizing and building your own runtime library*, page 124.

## PREBUILT RUNTIME LIBRARIES

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

- Core
- Data model
- Code model
- Size of the `double` floating-point type
- Library configuration—Normal or Full
- Position-independent code and read-only data (ROPI)

- Support for multithreading.

### To choose a library from the command line:



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

- Specify which library object file to use on the XLINK command line, like:  
`dllibname.r43`
- If you do not specify a library configuration, the default configuration will be used. However, you can specify the library configuration explicitly for the compiler:  
`--dlib_config C:\...\dllibname.h`

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

You can find the library object files in the subdirectory `430\lib\dlib` and the library configuration files in the `430\inc\dlib` subdirectory.

### Library filename syntax

The names of the libraries are constructed from these elements:

|                               |                                                                                                                                                                              |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>{lib}</code>            | is <code>d1</code> for the IAR DLIB runtime environment                                                                                                                      |
| <code>{core}</code>           | is either <code>430</code> or <code>430x</code>                                                                                                                              |
| <code>{code_model}</code>     | is empty for MSP430 devices. For MSP430X devices, it is one of <code>s</code> or <code>l</code> for the Small and Large code model, respectively.                            |
| <code>{data_model}</code>     | is empty for MSP430 devices. For MSP430X devices, it is one of <code>s</code> , <code>m</code> , or <code>l</code> for the Small, Medium, and Large data model, respectively |
| <code>{size_of_double}</code> | is either <code>f</code> for 32 bits or <code>d</code> for 64 bits                                                                                                           |
| <code>{lib_config}</code>     | is one of <code>n</code> or <code>f</code> for normal, and full, respectively.                                                                                               |
| <code>{ropi}</code>           | is <code>r</code> when position-independent code and read-only data is enabled, otherwise it is empty.                                                                       |
| <code>{threads}</code>        | is <code>t</code> when support for multithreading is enabled, otherwise empty.                                                                                               |

For example, the library `d1430xsmdfrt.r43` is configured for DLIB, the MSP430X core, the Small code model, the Medium data model, 64-bit double, the full library configuration, position-independent code and read-only data is enabled, and multithreading is enabled.



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

## FORMATTERS FOR PRINTF

The `printf` function uses a formatter called `_Printf`. The full version is quite large, and provides facilities not required in many embedded applications. To reduce the memory consumption, three smaller, alternative versions are also provided. Note that the `wprintf` variants are not affected.

This table summarizes the capabilities of the different formatters:

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

Table 12: Formatters for `printf`

† NoMb means without multibytes.

**Note:** It is possible to optimize these functions even further, but that requires that you rebuild the library. See *Configuration symbols for `printf` and `scanf`*, page 146.

The compiler can automatically detect which formatting capabilities are needed in a direct call to `printf`, if the formatting string is a string literal. This information is passed to the linker, which combines the information from all modules to select a suitable formatter for the application. However, if the formatting string is a variable, or if the call is indirect through a function pointer, the compiler cannot perform the analysis, forcing the linker to select the Full formatter. In this case, you might want to override the automatically selected `printf` formatter.



### To override the automatically selected `printf` formatter in the IDE:

- 1 Choose **Project>Options>General Options** to open the **Options** dialog box.
- 2 On the **Library Options** page, select the appropriate formatter.



### To override the automatically selected printf formatter from the command line:

- I Add one of these lines in the linker configuration file you are using:

```
-e_PrintfFull=_Printf
-e_PrintfFullNoMb=_Printf
-e_PrintfLarge=_Printf
-e_PrintfLargeNoMb=_Printf
_e_PrintfSmall=_Printf
-e_PrintfSmallNoMb=_Printf
-e_PrintfTiny=_Printf
-e_PrintfTinyNoMb=_Printf
```

## FORMATTERS FOR SCANF

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

This table summarizes the capabilities of the different formatters:

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

Table 13: Formatters for `scanf`

† NoMb means without multibytes.

**Note:** It is possible to optimize these functions even further, but that requires that you rebuild the library. See *Configuration symbols for printf and scanf*, page 146.

The compiler can automatically detect which formatting capabilities are needed in a direct call to `scanf`, if the formatting string is a string literal. This information is passed to the linker, which combines the information from all modules to select a suitable formatter for the application. However, if the formatting string is a variable, or if the call is indirect through a function pointer, the compiler cannot perform the analysis, forcing

the linker to select the full formatter. In this case, you might want to override the automatically selected `scanf` formatter.



### To manually specify the scanf formatter in the IDE:

- 1 Choose **Project>Options>General Options** to open the **Options** dialog box.
- 2 On the **Library Options** page, select the appropriate formatter.



### To manually specify the scanf formatter from the command line:

- 1 Add one of these lines in the linker configuration file you are using:

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

## THE C-SPY EMULATED I/O MECHANISM

The C-SPY emulated I/O mechanism works as follows:

- 1 The debugger will detect the presence of the function `__DebugBreak`, which will be part of the application if you linked it with the linker option for C-SPY emulated I/O.
- 2 In this case, the debugger will automatically set a breakpoint at the `__DebugBreak` function.
- 3 When your application calls a function in the DLIB low-level I/O interface, for example, `open`, the `__DebugBreak` function is called, which will cause the application to stop at the breakpoint and perform the necessary services.
- 4 The execution will then resume.

See also *Briefly about C-SPY emulated I/O*, page 118.

## MATH FUNCTIONS

Some C/C++ standard library math functions are available in different versions:

- The default versions
- Smaller versions (but less accurate)
- More accurate versions (but larger).

### Smaller versions

The functions `cos`, `exp`, `log`, `log2`, `log10`, `__iar_Log` (a help function for `log`, `log2`, and `log10`), `pow`, `sin`, `tan`, and `__iar_Sin` (a help function for `sin` and `cos`) exist in additional, smaller versions in the library. They are about 20% smaller and about 20% faster than the default versions. The functions handle INF and NaN values. The drawbacks are that they almost always lose some precision and they do not have the same input range as the default versions.

The names of the functions are constructed like:

```
__iar_xxx_small<f|l>
```

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



#### To specify which set of math functions to use:

Choose **Project>Options>General Options>Library Options>Math functions** and choose which set to use.

- 2 Link your application and the chosen set will be used.



#### To specify a complete set of math functions from the command line:

- 1 Use the `-f` option to extend the XLINK command line with the appropriate file:

| Extended command line file                 | Texas Instruments MathLib is used | double size |
|--------------------------------------------|-----------------------------------|-------------|
| <code>dlib_small_math.xcl</code>           | no                                | any         |
| <code>dlib_small_math_mathlib32.xcl</code> | yes                               | 32 bits     |
| <code>dlib_small_math_mathlib64.xcl</code> | yes                               | 64 bits     |

Table 14: Extended command line files for the small math functions

- 2 Link your application and the complete set will be used.



### To specify individual math functions from the command line:

Redirect the default function names to these names when linking, using these options:

```
-e __iar_sin_small=sin
-e __iar_cos_small=cos
-e __iar_tan_small=tan
-e __iar_log_small=log
-e __iar_log2_small=log2
-e __iar_log10_small=log10
-e __iar_exp_small=exp
-e __iar_pow_small=pow
-e __iar_Sin_small=__iar_Sin
-e __iar_Log_small=__iar_Log

-e __iar_sin_smallf=sinf
-e __iar_cos_smallf=cosf
-e __iar_tan_smallf=tanf
-e __iar_log_smallf=logf
-e __iar_log2_smallf=log2f
-e __iar_log10_smallf=log10f
-e __iar_exp_smallf=expf
-e __iar_pow_smallf=powf
-e __iar_Sin_smallf=__iar_Sinf
-e __iar_Log_smallf=__iar_Logf

-e __iar_sin_smalll=sinl
-e __iar_cos_smalll=cosl
-e __iar_tan_smalll=tanl
-e __iar_log_smalll=logl
-e __iar_log2_smalll=log2l
-e __iar_log10_smalll=log10l
-e __iar_exp_smalll=expl
-e __iar_pow_smalll=powl
-e __iar_Sin_smalll=__iar_Sinl
-e __iar_Log_smalll=__iar_Logl
```

Note that if you want to redirect any of the functions `sin`, `cos`, or `__iar_Sin`, you must redirect all three functions.

Note that if you want to redirect any of the functions `log`, `log2`, `log10`, or `__iar_Log`, you must redirect all four functions.

### More accurate versions

The functions `cos`, `pow`, `sin`, and `tan`, and the help functions `__iar_Sin` and `__iar_Pow` exist in versions in the library that are more exact and can handle larger

argument ranges. The drawback is that they are larger and slower than the default versions.

The names of the functions are constructed like:

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

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



### To specify individual math functions from the command line:

- I Redirect the default function names to these names when linking, using these options:

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

-e __iar_sin_accuratef=sinf
-e __iar_cos_accuratef=cosf
-e __iar_tan_accuratef=tanf
-e __iar_pow_accuratef=powf
-e __iar_Sin_accuratef=__iar_Sinf
-e __iar_Pow_accuratef=__iar_Powf

-e __iar_sin_accuratel=sinl
-e __iar_cos_accuratel=cosl
-e __iar_tan_accuratel=tanl
-e __iar_pow_accuratel=powl
-e __iar_Sin_accuratel=__iar_Sinl
-e __iar_Pow_accuratel=__iar_Powl
```

Note that if you want to redirect any of the functions `sin`, `cos`, or `__iar_Sin`, you must redirect all three functions.

Note that if you want to redirect any of the functions `pow` or `__iar_Pow`, you must redirect both functions.

## THE TEXAS INSTRUMENTS MATHLIB LIBRARY

Texas Instruments provides a library with a handful of hand-optimized 32-bit floating-point functions that use the hardware multiplier.

The MathLib library requires that the hardware multiplier is enabled. However, MathLib cannot be used in combination with *position-independent code and read-only data* (ROPI).



To use the MathLib library, choose **Project>Options>General Options>Library Configuration** and select **MathLib**.



To use the MathLib library on the command line, include the runtime library that corresponds to your device and project settings from `430\lib\MathLib`. Note that, to take effect, the MathLib library must be specified before the runtime library on the command line.



When you are using the MathLib library functions, interrupts are disabled while the hardware multiplier is used.

## SYSTEM STARTUP AND TERMINATION

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

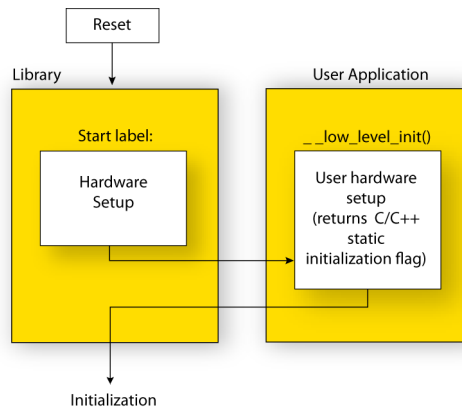
The code for handling startup and termination is located in the source files `cstartup.s43`, and `low_level_init.c` located in the `430\src\lib` directory.

For information about how to customize the system startup code, see *System initialization*, page 138.

### 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.

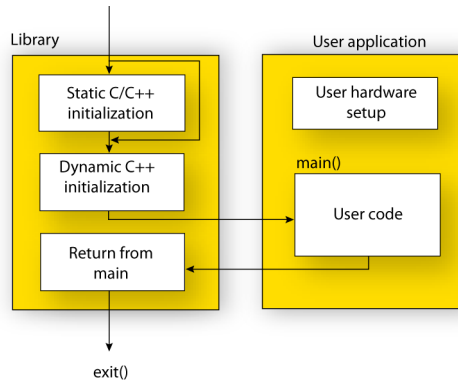
For the hardware initialization, it looks like this:



- When the CPU is reset it will start executing at the program entry label `__program_start` in the system startup code.

- The stack pointer (SP) is initialized
- If MPU support is included, the function `__iar_430_mpu_init` is called to initialize the MPU.
- 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:

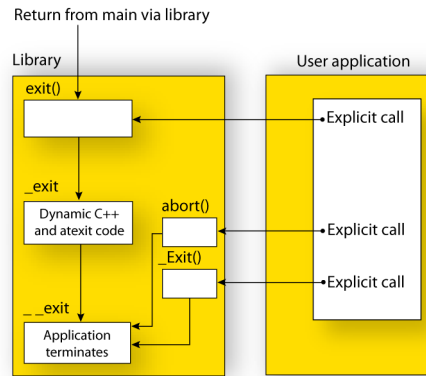


- The RAM area of `__ramfunc` functions is initialized
- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialization at system startup*, page 89.
- Static C++ objects are constructed
- The `main` function is called, which starts the application.



## System termination

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



An application can terminate normally in two different ways:

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

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

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

- 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 (and if using `atexit` is not sufficient), you can write your own implementation of the `__exit(int)` function.

The library files that you can override with your own versions are located in the `430\src\lib` directory. See *Overriding library modules*, page 123.

### C-SPY debugging support for system termination

If you have enabled C-SPY emulated I/O during linking, 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 emulate program termination. For more information, see *Briefly about C-SPY emulated I/O*, page 118.

## SYSTEM INITIALIZATION

It is likely that you need to adapt the 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 the system startup code.

You can do this by implementing your own version of the routine `__low_level_init`, which is called from the file `cstartup` before the data segments are initialized. Modifying the file `cstartup.s43` 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`.

**Note:** Regardless of whether you implement your own version of `__low_level_init` or the file `cstartup.s43`, you do not have to rebuild the library.

### Customizing `__low_level_init`

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 `cstartup` file

As noted earlier, you should not modify the file `cstartup.s43` if implementing your own 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 123.

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*.

## THE DLIB LOW-LEVEL I/O INTERFACE

The runtime library uses a set of low-level functions—which are referred to as the *DLIB low-level I/O interface*—to communicate with the target system. Most of the low-level functions have no implementation.

For more information about this, see *Briefly about input and output (I/O)*, page 116.

This table lists the functions in the DLIB low-level I/O interface:

| Function in DLIB low-level I/O interface      | Defined in source file                      |
|-----------------------------------------------|---------------------------------------------|
| <code>abort</code>                            | <code>abort.c</code>                        |
| <code>clock</code>                            | <code>clock.c</code>                        |
| <code>__close</code>                          | <code>close.c</code>                        |
| <code>__exit</code>                           | <code>xxexit.c</code>                       |
| <code>getenv</code>                           | <code>getenv.c</code>                       |
| <code>__getzone</code>                        | <code>getzone.c</code>                      |
| <code>__lseek</code>                          | <code>lseek.c</code>                        |
| <code>__open</code>                           | <code>open.c</code>                         |
| <code>raise</code>                            | <code>raise.c</code>                        |
| <code>__read</code>                           | <code>read.c</code>                         |
| <code>remove</code>                           | <code>remove.c</code>                       |
| <code>rename</code>                           | <code>rename.c</code>                       |
| <code>_ReportAssert</code>                    | <code>xreportassert.c</code>                |
| <code>signal</code>                           | <code>signal.c</code>                       |
| <code>system</code>                           | <code>system.c</code>                       |
| <code>__time32</code> , <code>__time64</code> | <code>time.c</code> , <code>time64.c</code> |
| <code>__write</code>                          | <code>write.c</code>                        |

Table 15: DLIB low-level I/O interface functions

**Note:** You should not use the low-level functions prefixed with `__` directly in your application. Instead you should use the standard library functions that use these functions. For example, to write to `stdout`, you should use standard library functions

like `printf` or `puts`, which in turn calls the low-level function `__write`. If you have forgot to implement a low-level function and your application will call that function via a standard library function, the linker will issues an error when you link in release build configuration.

**Note:** If you implement your own variants of the functions in this interface, your variants will be used even though you have enabled C-SPY emulated I/O, see *Briefly about C-SPY emulated I/O*, page 118.

## abort

|                        |                                                                                      |
|------------------------|--------------------------------------------------------------------------------------|
| Source file            | 430\src\lib\dlib\abort.c                                                             |
| Description            | Standard C library function that aborts execution.                                   |
| C-SPY debug action     | Notifies that the application has called abort.                                      |
| Default implementation | Calls <code>__exit(EXIT_FAILURE)</code> .                                            |
| See also               | <i>Briefly about retargeting</i> , page 118<br><i>System termination</i> , page 137. |

## clock

|                        |                                                                           |
|------------------------|---------------------------------------------------------------------------|
| Source file            | 430\src\lib\dlib\clock.c                                                  |
| Description            | Standard C library function that accesses the processor time.             |
| C-SPY debug action     | Returns the clock on the host computer.                                   |
| Default implementation | Returns <code>-1</code> to indicate that processor time is not available. |
| See also               | <i>Briefly about retargeting</i> , page 118.                              |

## \_\_close

|                    |                                                       |
|--------------------|-------------------------------------------------------|
| Source file        | 430\src\lib\dlib\close.c                              |
| Description        | Low-level function that closes a file.                |
| C-SPY debug action | Closes the associated host file on the host computer. |

Default implementation None.

See also *Briefly about retargeting*, page 118.

## **\_\_exit**

Source file 430\src\lib\dlib\xxexit.c

Description Low-level function that halts execution.

C-SPY debug action Notifies that the end of the application was reached.

Default implementation Loops forever.

See also *Briefly about retargeting*, page 118  
*System termination*, page 137.

## **getenv**

Source file 430\src\lib\dlib\getenv.c  
430\src\lib\dlib\environ.c

C-SPY debug action Accesses the host environment.

Default implementation The `getenv` function in the library 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.

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

See also *Briefly about retargeting*, page 118.

## **\_\_getzone**

|                        |                                                        |
|------------------------|--------------------------------------------------------|
| Source file            | 430\src\lib\dlib\getzone.c                             |
| Description            | Low-level function that returns the current time zone. |
| C-SPY debug action     | Not applicable.                                        |
| Default implementation | Returns " : ".                                         |
| See also               | <i>Briefly about retargeting</i> , page 118.           |

## **\_\_lseek**

|                        |                                                                                  |
|------------------------|----------------------------------------------------------------------------------|
| Source file            | 430\src\lib\dlib\lseek.c                                                         |
| Description            | Low-level function for changing the location of the next access in an open file. |
| C-SPY debug action     | Searches in the associated host file on the host computer.                       |
| Default implementation | None.                                                                            |
| See also               | <i>Briefly about retargeting</i> , page 118.                                     |

## **\_\_open**

|                        |                                              |
|------------------------|----------------------------------------------|
| Source file            | 430\src\lib\dlib\open.c                      |
| Description            | Low-level function that opens a file.        |
| C-SPY debug action     | Opens a file on the host computer.           |
| Default implementation | None.                                        |
| See also               | <i>Briefly about retargeting</i> , page 118. |

## raise

|                        |                                                                                                                |
|------------------------|----------------------------------------------------------------------------------------------------------------|
| Source file            | 430\src\lib\dlib\raise.c                                                                                       |
| Description            | Standard C library function that raises a signal.                                                              |
| C-SPY debug action     | Not applicable.                                                                                                |
| Default implementation | Calls the signal handler for the raised signal, or terminates with call to <code>__exit(EXIT_FAILURE)</code> . |
| See also               | <i>Briefly about retargeting</i> , page 118.                                                                   |

## \_\_read

|                        |                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Source file            | 430\src\lib\dlib\read.c                                                                                                                                                                                                                                                                                                                                                               |
| Description            | Low-level function that reads characters from <code>stdin</code> and from files.                                                                                                                                                                                                                                                                                                      |
| C-SPY debug action     | Directs <code>stdin</code> to the <b>Terminal I/O</b> window. All other files will read the associated host file.                                                                                                                                                                                                                                                                     |
| Default implementation | None.                                                                                                                                                                                                                                                                                                                                                                                 |
| Example                | <p>The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at :</p> <p>For information about the handles associated with the streams, see <i>Retargeting—Adapting for your target system</i>, page 122.</p> <p>For information about the @ operator, see <i>Controlling data and function placement in memory</i>, page 230.</p> |
| See also               | <i>Briefly about retargeting</i> , page 118.                                                                                                                                                                                                                                                                                                                                          |

## remove

|                    |                                                                 |
|--------------------|-----------------------------------------------------------------|
| Source file        | 430\src\lib\dlib\remove.c                                       |
| Description        | Standard C library function that removes a file.                |
| C-SPY debug action | Writes a message to the <b>Debug Log</b> window and returns -1. |

Default implementation Returns 0 to indicate success, but without removing a file.

See also *Briefly about retargeting*, page 118.

## rename

Source file 430\src\lib\dlib\rename.c

Description Standard C library function that renames a file.

C-SPY debug action None.

Default implementation Returns -1 to indicate failure.

See also *Briefly about retargeting*, page 118.

## **\_ReportAssert**

Source file 430\src\lib\dlib\xreportassert.c

Description Low-level function that handles a failed assert.

C-SPY debug action Notifies the C-SPY debugger about the failed assert.

Default implementation Failed asserts are reported by the function `__ReportAssert`. By default, it prints an error message and calls `abort`. If this is not the behavior you require, you can implement your own version of the function.

The assert macro is defined in the header file `assert.h`. To turn off assertions, define the symbol `NDEBUG`.



In the IDE, the 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 367.

See also *Briefly about retargeting*, page 118.

## signal

Source file 430\src\lib\dlib\signal.c

Description Standard C library function that changes signal handlers.



|                        |                                                                                                                                   |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| C-SPY debug action     | Not applicable.                                                                                                                   |
| Default implementation | As specified by Standard C. You might want to modify this behavior if the environment supports some kind of asynchronous signals. |
| See also               | <i>Briefly about retargeting</i> , page 118.                                                                                      |

## system

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Source file            | 430\src\lib\dlib\system.c                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description            | Standard C library function that executes commands.                                                                                                                                                                                                                                                                                                                                                                                                          |
| C-SPY debug action     | Notifies the C-SPY debugger that <code>system</code> has been called and then returns <code>-1</code> .                                                                                                                                                                                                                                                                                                                                                      |
| Default implementation | If you need to use the <code>system</code> function, you must implement it yourself. The <code>system</code> function available in the library returns <code>0</code> if a null pointer is passed to it to indicate that there is no command processor; otherwise it returns <code>-1</code> to indicate failure. If this is not the functionality that you require, you can implement your own version. This does not require that you rebuild the library. |
| See also               | <i>Briefly about retargeting</i> , page 118.                                                                                                                                                                                                                                                                                                                                                                                                                 |

## \_\_time32, \_\_time64

|                        |                                                                          |
|------------------------|--------------------------------------------------------------------------|
| Source file            | 430\src\lib\dlib\time.c<br>430\src\lib\dlib\time64.c                     |
| Description            | Low-level functions that return the current calendar time.               |
| C-SPY debug action     | Returns the time on the host computer.                                   |
| Default implementation | Returns <code>-1</code> to indicate that calendar time is not available. |
| See also               | <i>Briefly about retargeting</i> , page 118.                             |

## \_\_write

|             |                                                                                          |
|-------------|------------------------------------------------------------------------------------------|
| Source file | 430\src\lib\dlib\write.c                                                                 |
| Description | Low-level function that writes to <code>stdout</code> , <code>stderr</code> , or a file. |

|                        |                                                                                                                                                                                                                                                                       |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C-SPY debug action     | Directs <code>stdout</code> and <code>stderr</code> to the <b>Terminal I/O</b> window. All other files will write to the associated host file.                                                                                                                        |
| Default implementation | None.                                                                                                                                                                                                                                                                 |
| Example                | The code in this example uses memory-mapped I/O to write to an LCD display, whose port is assumed to be located at address :<br><br>For information about the handles associated with the streams, see <i>Retargeting—Adapting for your target system</i> , page 122. |
| See also               | <i>Briefly about retargeting</i> , page 118.                                                                                                                                                                                                                          |

## CONFIGURATION SYMBOLS FOR PRINTF AND SCANF

If the provided formatters do not meet your requirements (*Formatters for printf*, page 129 and *Formatters for scanf*, page 130), you can customize the Full formatters. Note that this means you must rebuild the runtime library.

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

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

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

Table 16: Descriptions of printf configuration symbols

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

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

Table 17: Descriptions of `scanf` configuration symbols

### To customize the formatting capabilities, you must;

- 1 Define the configuration symbols in the library configuration file according to your application requirements.
- 2 Rebuild the library, see *Customizing and building your own runtime library*, page 124.

## CONFIGURATION SYMBOLS FOR FILE INPUT AND OUTPUT

File I/O is only supported by libraries with the Full library configuration, see *Runtime library configurations*, page 126, or in a customized library when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is defined. If this symbol is not defined, functions taking a `FILE *` argument cannot be used.

To customize your library and rebuild it, see *Customizing and building your own runtime library*, page 124.

## 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 which library configuration 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. See *Runtime library configurations*, page 126.

The DLIB runtime 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 and rebuilding the library

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

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

### Locale configuration symbols

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

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

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Customizing and building your own runtime library*, page 124.

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

### Building a library without support for locale interface

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

### Building a library with support for locale interface

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

### Changing locales at runtime

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

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

*lang\_REGION*

or

*lang\_REGION.encoding*

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

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

### Example

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

```
setlocale (LC_ALL, "sv_FI.Utf8");
```

## STR TOD

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 *Customizing and building your own runtime library*, page 124.

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

| Linker configuration file | Type of hardware multiplier |
|---------------------------|-----------------------------|
| multiplier.xcl            | 16 or 16s                   |
| multiplier32.xcl          | 32                          |

Table 18: Additional linker configuration files for the hardware multiplier

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

## MPU/IPE SUPPORT

### Memory Protection Unit (MPU)

Some MSP430 devices with FRAM contain a Memory Protection Unit (MPU) that allows the FRAM memory to be divided into ranges with different access rights. The runtime library provides support for automatically initializing the MPU with ranges based on the content and placement of code and data in your application.

The linker configuration files for devices with an MPU group the segments in FRAM such that three zones with different access requirements can be defined. The borders between these zones are marked by the segments `MPU_B1` and `MPU_B2`, respectively. Persistent data, and the heaps for dynamically allocated data, are placed in the low zone,

with *read-write* permissions. For devices with IPE support, the middle zone is dedicated to IPE code and data, with *read-write-execute* permissions. Constants and code are then placed in the high zone, with *read-execute* permissions. For devices without IPE support, the middle zone is instead given *read-only* permissions, and holds constants.



To include the MPU initialization routine, and adapt the memory placement to the alignment requirements of the MPU, select **Project>Options>General Options>MPU/IPE>Support MPU**. To actually enable the MPU, you must also select **Project>Options>General Options>MPU/IPE>Enable MPU**. Use the other options on that page to adapt the MPU initialization to your application needs, including setting specific permissions on the information memory.



From the command line, send one of the following options to the linker to include the MPU support:

```
-g?mpu1_4k_init For devices with MPU but no IPE, and 4k memory
-g?mpu1_8k_init For devices with MPU but no IPE, and 8k memory
-g?mpu1_16k_init For devices with MPU but no IPE, and 16k memory
-g?mpu2_init For devices with both MPU and IPE
```

You must also define the hexadecimal values that will be put in the MPUSAM and MPUCTL0 registers, respectively, using for example:

```
-D__iar_430_MPUSAM_value=1573
-D__iar_430_MPUCTL0_value=0001
```

If the automatic setup for the MPU does not suit your application, you can manually override the module `?MpuInit` with your own implementation, without rebuilding the runtime library.

## Intellectual Property Encapsulation (IPE)

Some MSP430 devices with MPU support also contain the Intellectual Property Encapsulation functionality. When enabled, this prevents a certain range of memory to be read by a debugger. Only code located within the IPE range can read the data.

The runtime library provides support for automatically initializing the IPE to protect the data and code in certain predefined memory segments. If IPE support is included, special data structures are placed in the `IPESIGNATURE` memory, that are read by the boot code to initialize the IPE functionality.

The linker configuration files for devices with IPE functionality define the segments `IPEDATA16_N`, `IPEDATA16_C` and `IPECODE16` in FRAM, surrounded by the border segments `IPE_B1` and `IPE_B2`. Make sure to place writable `__no_init` data to be protected in the `IPEDATA16_N` segment, constant data in `IPEDATA16_C`, and the code to read the data in `IPECODE16`. Additionally, any user-defined segment linked between the border segments `IPE_B1` and `IPE_B2` will be protected.

For more information about how to place code and data in specific segments, see Controlling data and function placement in memory, page 230.



To include the IPE initialization data structures, and enforce the alignment requirements of the IPE, select **Project>Options>General Options>MPU/IPE>Support IPE**. To actually enable the IPE, also select **Project>Options>General Options>MPU/IPE>Enable IPE**. Use the other options on that page to further adapt the IPE initialization to your application needs.



From the command line, send the following option to the linker to include the IPE support:

```
-g_iar_430_ipe_signature
```

You must also define the hexadecimal value that will be put in the MPUIPC0 register, using for example:

```
-D_iar_430_MPUIPC0_value=0040
```

If the automatic setup for the IPE does not suit your application, you can manually override the module `?IpeInit` with your own implementation, without rebuilding the runtime library.

**Note:** Once IPE has been enabled on a device, the IPE area will remain locked until cleared.



To clear the IPE area when downloading an application, select **Project>Options>Debugger>FET Debugger>Download>Erase main and Information memory inc. IP PROTECTED area**.

---

## Managing a multithreaded environment

In a multithreaded environment, the standard library must treat all library objects according to whether they are global or local to a thread. If an object is a true global object, any updates of its state must be guarded by a locking mechanism to make sure that only one thread can update it at any given time. If an object is local to a thread, the static variables containing the object state must reside in a variable area local to that thread. This area is commonly named *thread-local storage* (TLS).

The low-level implementations of locks and TLS are system-specific, and is not included in the DLIB runtime environment. If you are using an RTOS, check if it provides some or all of the required functions. Otherwise, you must provide your own.



## MULTITHREAD SUPPORT IN THE DLIB RUNTIME ENVIRONMENT

The DLIB runtime environment uses two kinds of locks—*system locks* and *file stream locks*. The file stream locks are used as guards when the state of a file stream is updated, and are only needed in the Full library configuration. The following objects are guarded with system locks:

- The heap (in other words when `malloc`, `new`, `free`, `delete`, `realloc`, or `calloc` is used).
- The file system (only available in the Full library configuration), but not the file streams themselves. The file system is updated when a stream is opened or closed, in other words when `fopen`, `fclose`, `fdopen`, `fflush`, or `freopen` is used.
- The signal system (in other words when `signal` is used).
- The temporary file system (in other words when `tmpnam` is used).
- Dynamically initialized function local objects with static storage duration.

These library objects use TLS:

| Library objects using TLS | When these functions are used                                                                                                                                    |
|---------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Error functions           | <code>errno</code> , <code>strerror</code>                                                                                                                       |
| Locale functions          | <code>localeconv</code> , <code>setlocale</code>                                                                                                                 |
| Time functions            | <code>asctime</code> , <code>localtime</code> , <code>gmtime</code> , <code>mktime</code>                                                                        |
| Multibyte functions       | <code>mbrlen</code> , <code>mbrtowc</code> , <code>mbsrtowc</code> , <code>mbtowc</code> , <code>wcrtomb</code> ,<br><code>wcsrtomb</code> , <code>wctomb</code> |
| Rand functions            | <code>rand</code> , <code>srand</code>                                                                                                                           |
| Miscellaneous functions   | <code>atexit</code> , <code>strtok</code>                                                                                                                        |

Table 19: Library objects using TLS

**Note:** If you are using `printf/scanf` (or any variants) with formatters, each individual formatter will be guarded, but the complete `printf/scanf` invocation will not be guarded.

If one of the C++ variants is used together with the DLIB runtime environment with multithread support, the compiler option `--guard_calls` must be used to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.

## ENABLING MULTITHREAD SUPPORT

Prebuilt libraries with multithread support are provided in the product installation. There are no prebuilt runtime libraries with support for multithreading, which means you must first customize the library by adding the line `#define _DLIB_THREAD_SUPPORT 3` in

the library configuration file and rebuild your library. See also, *Customizing and building your own runtime library*, page 124.

### To configure multithread support for use with threaded applications:

1 To enable multithread support:



On the command line, use the linker option `--threaded_lib` and use one of the prebuilt libraries with thread support (`d1430*t.r43`) or your own customized library.



In the IDE, choose **Project>Options>General Options>Library**

**Configuration>Enable thread support in the library.** The linker option `--threaded_lib` and the matching prebuilt library with thread support will automatically be used. If one of the C++ variants is used, the IDE will automatically use the compiler option `--guard_calls` to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.

2 To complement the built-in multithread support in the runtime library, you must also:

- Implement code for the library's system locks interface, see *System locks interface*, page 155.
- If file streams are used, implement code for the library's file stream locks interface or redirect the interface to the system locks interface (using the linker option `-e`), see *File streams locks interface*, page 155.
- Implement code that handles thread creation, thread destruction, and TLS access methods for the library, see *TLS handling*, page 155.

See also *Lock usage*, page 155 for information about used file streams locks and system locks.

You can find the required declaration of functions and definitions of macros in the `DLib_Threads.h` file, which is included by `yvals.h`.

3 Build your project.

**Note:** If you are using a third-party RTOS, check their guidelines for how to enable multithread support with IAR Systems tools.

## System locks interface

This interface must be fully implemented for system locks to work:

```
typedef void *__iar_Rmtx; /* Lock info object */

void __iar_system_Mtxinit(__iar_Rmtx *); /* Initialize a system
 lock */
void __iar_system_Mtxdst(__iar_Rmtx *); /* Destroy a system lock */
void __iar_system_Mtxlock(__iar_Rmtx *); /* Lock a system lock */
void __iar_system_Mtxunlock(__iar_Rmtx *); /* Unlock a system
 lock */
```

The lock and unlock implementation must survive nested calls.

## File streams locks interface

This interface is only needed for the Full library configuration. If file streams are used, they can either be fully implemented or they can be redirected to the system locks interface. This interface must be implemented for file streams locks to work:

```
typedef void *__iar_Rmtx; /* Lock info object */

void __iar_file_Mtxinit(__iar_Rmtx *); /* Initialize a file lock */
void __iar_file_Mtxdst(__iar_Rmtx *); /* Destroy a file lock */
void __iar_file_Mtxlock(__iar_Rmtx *); /* Lock a file lock */
void __iar_file_Mtxunlock(__iar_Rmtx *); /* Unlock a file lock */
```

The lock and unlock implementation must survive nested calls.

## Lock usage

The number of locks that the DLIB runtime environment assumes exist are:

- `_FOPEN_MAX`—the maximum number of file stream locks. These locks are only used in the Full library configuration, in other words only if both the macro symbols `_DLIB_FILE_DESCRIPTOR` and `_FILE_OP_LOCKS` are true.
- `_MAX_LOCK`—the maximum number of system locks.

Note that even if the application uses fewer locks, the runtime environment will initialize and destroy all the locks above.

For information about the initialization and destruction code, see `xsyslock.c`.

## TLS handling

The DLIB runtime environment supports TLS memory areas for two types of threads: the *main thread* (the `main` function including the system startup and exit code) and *secondary threads*.

The main thread's TLS memory area:

- Is automatically created and initialized by your application's startup sequence
- Is automatically destructed by the application's destruct sequence
- Is located in the segment `TLS16_I`
- Exists also for non-threaded applications.

Each secondary thread's TLS memory area:

- Must be manually created and initialized
- Must be manually destructed
- Is located in a manually allocated memory area.

If you need the runtime library to support secondary threads, you must override the function:

```
void *__iar_dlib_perthread_access(void *symp);
```

The parameter is the address to the TLS variable to be accessed—in the main thread's TLS area—and it should return the address to the symbol in the current TLS area.

Two interfaces can be used for creating and destroying secondary threads. You can use the following interface that allocates a memory area on the heap and initializes it. At deallocation, it destroys the objects in the area and then frees the memory.

```
void *__iar_dlib_perthread_allocate(void);
void __iar_dlib_perthread_deallocate(void *);
```

Alternatively, if the application handles the TLS allocation, you can use this interface for initializing and destroying the objects in the memory area:

```
void __iar_dlib_perthread_initialize(void *);
void __iar_dlib_perthread_destroy(void *);
```

These macros can be helpful when you implement an interface for creating and destroying secondary threads:

| Macro                                                      | Description                                                                                                                                                |
|------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__IAR_DLIB_PERTHREAD_SIZE</code>                     | The size needed for the TLS memory area.                                                                                                                   |
| <code>__IAR_DLIB_PERTHREAD_INIT_SIZE</code>                | The initializer size for the TLS memory area. You should initialize the rest of the TLS memory area, up to <code>__IAR_DLIB_PERTHREAD_SIZE</code> to zero. |
| <code>__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(symbolptr)</code> | The offset to the symbol in the TLS memory area.                                                                                                           |

Table 20: Macros for implementing TLS allocation

Note that the size needed for TLS variables depends on which resources in the DLIB runtime environment your application uses.

This is an example of how you can handle threads:

```
#include <yvals.h>

/* A thread's TLS pointer */
void _DLIB_TLS_MEMORY *TLSp;

/* Are we in a secondary thread? */
int InSecondaryThread = 0;

/* Allocate a thread-local TLS memory
 area and store a pointer to it in TLSp. */
void AllocateTLS()
{
 TLSp = __iar_dlib_perthread_allocate();
}

/* Deallocate the thread-local TLS memory area. */
void DeallocateTLS()
{
 __iar_dlib_perthread_deallocate(TLSp);
}

/* Access an object in the
 thread-local TLS memory area. */
void _DLIB_TLS_MEMORY *__iar_dlib_perthread_access(
 void _DLIB_TLS_MEMORY *sympb)
{
 char _DLIB_TLS_MEMORY *p = 0;
 if (InSecondaryThread)
 p = (char _DLIB_TLS_MEMORY *) TLSp;
 else
 p = (char _DLIB_TLS_MEMORY *)
 __segment_begin("__DLIB_PERTHREAD");

 p += __IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(sympb);
 return (void _DLIB_TLS_MEMORY *) p;
}
```

The `TLSp` variable is unique for each thread, and must be exchanged by the RTOS or manually whenever a thread switch occurs.

## **TLS IN THE LINKER CONFIGURATION FILE**

If threads are used, the main thread's TLS memory area must be initialized by plain copying because the initializers are used for each secondary thread's TLS memory area as well. This is controlled by the following statement in your linker configuration file:

```
-QTLS16_I=TLS16_ID
```

Both the `TLS16_I` segment and the `TLS16_ID` segment must be placed in default memory for RAM and ROM, respectively.

The startup code will copy `TLS16_ID` to `TLS16_I`.

# The CLIB runtime environment

- Using a prebuilt runtime library
- Input and output
- System startup and termination
- Overriding default library modules
- Customizing system initialization
- C-SPY emulated I/O

Note that the CLIB runtime environment does not support any C99 functionality. For example, complex numbers and variable length arrays are not supported. Neither does CLIB support C++.

Note that the legacy CLIB runtime environment is provided for backward compatibility and should not be used for new application projects. Among other things, it does not support ROPI nor multithreading.

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

---

## Using a prebuilt runtime library

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

- Core
- Code model
- Data model
- Size of the `double` floating-point type.

## CHOOSING A RUNTIME LIBRARY

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

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

```
cllibname.r43
```

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

## RUNTIME LIBRARY FILENAME SYNTAX

The runtime library names are constructed in this way:

```
{lib}{core}{code_model}{data_model}{size_of_double}.r43
```

where

- `{lib}` `c1` for the CLIB runtime library
- `{core}` is either `430` or `430X`
- `{code_model}` is empty for MSP430 devices. For MSP430X devices, it is one of `s` or `l`, for the Small and Large code model, respectively
- `{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.

---

## Input and output

You can customize:

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

### CHARACTER-BASED I/O

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



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

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

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

```
__no_init volatile unsigned char devIO @ 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 123.

## FORMATTERS USED BY PRINTF AND SPRINTF

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

```
_large_write
_medium_write
_small_write
```

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

### **`_large_write`**

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

### **`_medium_write`**

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

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

`_medium_write` is considerably smaller than the large version.

### `_small_write`

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



### Specifying the printf formatter in the IDE

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



### Specifying the printf formatter from the command line

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

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

### Customizing printf

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

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 123.

### FORMATTERS USED BY SCANF AND SSCANF

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

```
_large_read
_medium_read
```

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

## **`_large_read`**

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

## **`_medium_read`**

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



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

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



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

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

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

---

## **System startup and termination**

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

The code for handling startup and termination is located in the source files `cstartup.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`.

### **SYSTEM STARTUP**

When an application is initialized, several steps are performed:

- If MPU support is included, the function `__iar_430_mpu_init` is called to initialize the MPU.
- The custom function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables.

- The `main` function is called, which starts the application.

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

## SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

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

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

---

## Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 123, in the chapter *The DLIB runtime environment*.

---

## Customizing system initialization

For information about how to customize system initialization, see *Customizing `__low_level_init`*, page 138.

---

## C-SPY emulated I/O

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

## THE DEBUGGER TERMINAL I/O WINDOW

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

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

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

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

## TERMINATION

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

---

## Hardware multiplier support

Some MSP430 devices contain a hardware multiplier. For information about how the compiler supports this unit, see *Hardware multiplier support*, page 150.

---

## MPU/IPE support

Some MSP430 devices support Memory Protection Unit (MPU) and Intellectual Property Encapsulation, see *MPU/IPE support*, page 150.



# Assembler language interface

- Mixing C and assembler
- Calling assembler routines from C
- Calling assembler routines from C++
- Calling convention
- Assembler instructions used for calling functions
- Memory access methods
- Call frame information

---

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

## MIXING C AND ASSEMBLER MODULES

It is possible to write parts of your application in assembler and mix them with your C or C++ modules.

This gives several benefits compared to using inline assembler:

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

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

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

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

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

The first question is discussed in the section *Calling assembler routines from C*, page 175. The following two are covered in the section *Calling convention*, page 179.

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 189.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 175, and *Calling assembler routines from C++*, page 178, respectively.

## INLINE ASSEMBLER

Inline assembler can be used for inserting assembler instructions directly into a C or C++ function. Typically, this can be useful if you need to:

- Access hardware resources that are not accessible in C (in other words, when there is no definition for an SFR or there is no suitable intrinsic function available).
- Manually write a time-critical sequence of code that if written in C will not have the right timing.
- Manually write a speed-critical sequence of code that if written in C will be too slow.
- Call functions using a non-standard calling convention.

An inline assembler statement is similar to a C function in that it can take input arguments (input operands), have return values (output operands), and read or write to C symbols (via the operands). An inline assembler statement can also declare *clobbered resources* (that is, values in registers and memory that have been overwritten).

### Limitations

Most things you can do in normal assembler language are also possible with inline assembler, with the following differences:

- In general, assembler directives will cause errors or have no meaning. However, data definition directives will work as expected.
- Resources used (registers, memory, etc) that are also used by the C compiler must be declared as operands or clobbered resources.
- If you do not want to risk that the inline assembler statement to be optimized away by the compiler, you must declare it `volatile`.
- Accessing a C symbol or using a constant expression requires the use of operands.
- Dependencies between the expressions for the operands might result in an error.

### Risks with inline assembler

Without operands and clobbered resources, inline assembler statements have no interface with the surrounding C source code. This makes the inline assembler code fragile, and might also become a maintenance problem if you update the compiler in the

future. There are also several limitations to using inline assembler without operands and clobbered resources:

- The inline assembler statement will be *volatile* and *clobbered memory* is not implied. This means that the compiler will not remove the assembler statement. It will simply be inserted at the given location in the program flow. The consequences or side-effects that 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.



The following example demonstrates the risks of using the `asm` keyword without operands and clobbers:

```
int Add(int term1, int term2)
{
 asm("add.w R13, R12");
 return term1;
}
```

In this example:

- The function `Add` assumes that values are passed and returned in registers in a way that they might not always be, for example if the function is inlined.
- The `ADD` instruction updates the condition flags, which you specify using the `cc` clobber operand. Otherwise, the compiler will assume that the condition flags are not modified.

Inline assembler without using operands or clobbered resources is therefore often best avoided.

## Reference information for inline assembler

The `asm` and `__asm` keywords both insert inline 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.

### Syntax

The syntax of an inline assembler statement is (similar to the one used by GNU `gcc`):

```
asm [volatile](string [assembler-interface])
```

*string* can contain one or more valid assembler instructions or data definition assembler directives, separated by `\n`.

For example:

```
void LoopForever(void)
{
 asm("label: nop\n"
 " jmp label");
}
```

Note that any labels you define in the inline assembler statement will be local to that statement. You can use this for loops or conditional code.

If you define a label in an inline assembler statement using two colons (for example: "*label::* nop\n") instead of one, the label will be public, not only in the inline assembler statement, but in the module as well. This feature is intended for testing only.

*assembler-interface* is:

```
: comma-separated list of output operands /* optional */
: comma-separated list of input operands /* optional */
: comma-separated list of clobbered resources /* optional */
```

## Operands

An inline assembler statement can have one input and one output comma-separated list of operands. Each operand consists of an optional symbolic name in brackets, a quoted constraint, followed by a C expression in parentheses.

## Syntax of operands

```
[[symbolic-name]] "[modifiers]constraint" (expr)
```

For example:

```
int Add(int term1, int term2)
{
 int sum;

 asm("mov %1,%0\n"
 "add %2,%0"
 : "=r"(sum)
 : "r"(term1), "r"(term2));

 return sum;
}
```

In this example, the assembler instruction uses one output operand, *sum*, two input operands, *term1* and *term2*, and no clobbered resources.

It is possible to omit any list by leaving it empty. For example:

```
char matrix[M][N];

void MatrixSetBit(int row)
{
 asm("BIS.W #1,0(%0)" : : "r" (&matrix[row][0]) : "memory");
}
```

Operand constraints

| Constraint | Description                                               |
|------------|-----------------------------------------------------------|
| r          | Uses a general purpose register or group of registers     |
| R4-R15     | A specific general purpose register or group of registers |
| i          | Integer constant (not valid as output)                    |
| m          | Operand referring to a memory location                    |

Table 21: Inline assembler operand constraints

Constraint modifiers

Constraint modifiers can be used together with a constraint to modify its meaning. This table lists the supported constraint modifiers:

| Modifier | Description                                                                                                   |
|----------|---------------------------------------------------------------------------------------------------------------|
| =        | Write-only operand                                                                                            |
| +        | Read-write operand                                                                                            |
| &        | Early clobber output operand which is written to before the instruction has processed all the input operands. |

Table 22: Supported constraint modifiers

Referring to operands

Assembler instructions refer to operands by prefixing their order number with %. The first operand has order number 0 and is referred to by %0.

If the operand has a symbolic name, you can refer to it using the syntax %[operand.name]. Symbolic operand names are in a separate namespace from C/C++ code and can be the same as a C/C++ variable names. Each operand name must however be unique in each assembler statement. For example:

```

int Add(int term1, int term2)
{
 int sum;

 asm("mov %[Rm], %[Rd]\n"
 "add %[Rn], %[Rd]"
 : [Rd]"=r"(sum)
 : [Rn]"r"(term1), [Rm]"r"(term2));
 return sum;
}

```

- Input operands** Input operands cannot have any constraint modifiers, but they can have any valid C expression as long as the type of the expression fits the register.
- The C expression will be evaluated just before any of the assembler instructions in the inline assembler statement and assigned to the constraint, for example a register.
- Output operands** Output operands must have = as a constraint modifier and the C expression must be an l-value and specify a writable location. For example, =r for a write-only general purpose register. The constraint will be assigned to the evaluated C expression (as an l-value) immediately after the last assembler instruction in the inline assembler statement. Input operands are assumed to be consumed before output is produced and the compiler may use the same register for an input and output operand. To prohibit this, prefix the output constraint with & to make it an early clobber resource, for example =&r. This will ensure that the output operand will be allocated in a different register than the input operands.
- Input/output operands** An operand that should be used both for input and output must be listed as an output operand and have the + modifier. The C expression must be an l-value and specify a writable location. The location will be read immediately before any assembler instructions and it will be written to right after the last assembler instruction.
- This is an example of using a read-write operand:
- ```

int Double(int value)
{
    asm("add.w %0,%0" : "+r"(value));
    return value;
}

```
- In the example above, the input value for `value` will be placed in a general purpose register. After the assembler statement, the result from the `ADD` instruction will be placed in the same register.
- Clobbered resources** An inline assembler statement can have a list of clobbered resources.
- ```

"resource1", "resource2", ...

```

Specify clobbered resources to inform the compiler about which resources the inline assembler statement destroys. Any value that resides in a clobbered resource and that is needed after the inline assembler statement will be reloaded.

Clobbered resources will not be used as input or output operands.

This is an example of how to use clobbered resources:

```
int Add(int term1, int term2)
{
 int sum;

 asm("mov %1,%0\n"
 "add %2,%0"
 : "=r" (sum)
 : "r" (term1), "r" (term2)
 : "cc");

 return sum;
}
```

In this example the condition codes will be modified by the `ADD` instruction. Therefore, `"cc"` must be listed in the clobber list.

This table lists valid clobbered resources:

| <b>Clobber</b> | <b>Description</b>                                                                                                                                 |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| R4-R15         | General purpose register                                                                                                                           |
| cc             | The condition flags (N, Z, V, and C)                                                                                                               |
| memory         | To be used if the instructions modify any memory. This will avoid keeping memory values cached in registers across the inline assembler statement. |

*Table 23: List of valid clobbers*

## Operand modifiers

An operand modifier is a single letter between the % and the operand number, which is used for transforming the operand.

In the example below, the modifiers A and B are used for accessing the least and most significant 16 bits, respectively, of the operands:

```
unsigned long Mov32(void)
{
 unsigned long sum;

 asm("mov.w %A1,%A0\n"
 "mov.w %B1,%B0"
 : "=r"(sum)
 : "i"(0x12345678UL));
 return sum;
}
```

This table describes the transformation performed by each valid modifier:

| Modifier | Description                                                      |
|----------|------------------------------------------------------------------|
| A        | The first (lowest) 16 bits of the operand.                       |
| B        | The second 16 bits of the operand.                               |
| C        | The third 16 bits of the operand. (64-bit types only)            |
| D        | The fourth (highest) 16 bits of the operand. (64-bit types only) |

Table 24: Operand modifiers and transformations

## AN EXAMPLE OF HOW TO USE CLOBBERED MEMORY

```
void MemSet(char * memory, char value)
{
 asm("mov.w %0, R8\n"
 "mov.b %1, 0(R8)"
 : /* No output constraints. */
 : "r"(memory), "r"(value)
 : "memory", "R8");
}
```

## Calling assembler routines from C

An assembler routine that will be called from C must:

- Conform to the calling convention
- Have a PUBLIC entry-point label

- Be declared as external before any call, to allow type checking and optional promotion of parameters, as in these examples:

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

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

## CREATING SKELETON CODE

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

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

```
extern int gInt;
extern char gChar;

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

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

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

## COMPILING THE SKELETON 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 . -On -e
```

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`. The `-On` option means that no optimization will be used and `-e` enables language extensions. In addition, make sure to use relevant compiler options, usually the same as you use for other C or C++ source files in your project.

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
- 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 189.

---

## Calling assembler routines from C++

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

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

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

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

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

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

```
class MyClass;

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

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

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

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

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

---

## Calling convention

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

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

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:

```
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 176.

You should also specify a value to the runtime model attribute `__rt_version` using the `RTMODEL` assembler directive:

```
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.

## FUNCTION DECLARATIONS

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

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

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

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

## USING C LINKAGE IN C++ SOURCE CODE

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

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

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

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

```
#ifndef __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 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. Because the code generated by the compiler in the Small data model does not use the upper four bits of registers, it is only necessary to save and restore these bits if they are used by any assembler routines.

**Note:** When compiling using the options `--lock_r4` or `--lock_r5`, the R4 and R5 registers are not used.

## FUNCTION ENTRANCE

Parameters can be passed to a function using one of these basic methods:

- In registers
- 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:

| Parameters     | Passed in registers, Version1 | Passed in registers, Version2 |
|----------------|-------------------------------|-------------------------------|
| 8-bit values   | R12, R14                      | R12 to R15                    |
| 16-bit values  | R12, R14                      | R12 to R15                    |
| 20-bit values* | R12, R14                      | R12 to R15                    |
| 32-bit values  | (R13:R12), (R15:R14)          | (R13:R12), (R15:R14)          |

*Table 25: Registers used for passing parameters*

| Parameters    | Passed in registers, Version 1        | Passed in registers, Version 2        |
|---------------|---------------------------------------|---------------------------------------|
| 64-bit values | (R15:R14:R13:R12),<br>(R11:R10:R9:R8) | (R15:R14:R13:R12),<br>(R11:R10:R9:R8) |

Table 25: Registers used for passing parameters (Continued)

\* On MSP430X devices, if the Small data model is combined with the Large code model, function pointers are passed as 32-bit values.

The assignment of registers to parameters is a straightforward process.

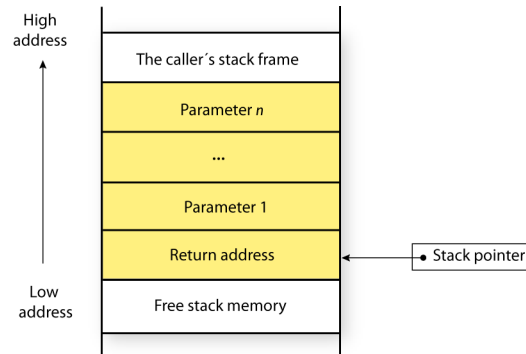
For Version 1, the first parameter is assigned to R12 or R13:R12, depending on the size of the parameter. The second parameter is passed in R14 or R15:R14. Should there be no more available registers, the parameter is passed on the stack.

For Version 2, 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.

This figure illustrates how parameters are stored on the stack:



## FUNCTION EXIT

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

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

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

### Registers used for returning values

These registers are available for returning values:

| Return values  | Passed in registers     |
|----------------|-------------------------|
| 8-bit values   | R12                     |
| 16-bit values  | R12                     |
| 20-bit values* | R12                     |
| 32-bit values  | (R13 : R12)             |
| 64-bit values  | (R15 : R14 : R13 : R12) |

Table 26: Registers used for returning values

\* On MSP430X devices, if the Small data model is combined with the Large code model, function pointers are returned as 32-bit values.

### Stack layout at function exit

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

### RESTRICTIONS FOR SPECIAL FUNCTION TYPES

For information about how the `__interrupt`, `__raw`, and `__save_reg20` keywords affect the calling convention, see *Interrupt functions*, page 73.

### 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
ret
```

## Example 2

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

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

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

The calling function must reserve 10 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register `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 MyStruct`.

```
struct MyStruct
{
 int mA[20];
};
```

```
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:

```
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`.

---

## 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.

Functions can be called in different ways—directly or via a function pointer. In this section we will discuss how these types of calls will be performed for each core and code model.

### NORMAL (NON-ROPI) CODE

If the `--ropi` option is not specified, the following instruction sequences are used for calling and returning from functions.

#### MSP430 and MSP430X using the Small code model

For the MSP430 core, and for the MSP430X core when the Small code model is used, functions are called with the `CALL` instruction, and returned from with the `RET` instruction.

The return address is stored on the stack and uses 2 bytes of memory.

#### MSP430X using the Large code model

For the MSP430X core when the Large code model is used, functions are called with the `CALLA` instruction, and returned from with the `RETA` instruction.

The return address is stored on the stack and uses 4 bytes of memory.

### POSITION-INDEPENDENT CODE AND READ-ONLY DATA

When the `--ropi` option is specified, the following instruction sequences are used for calling and returning from functions.

#### MSP430 and MSP430X using the Small code model

For the MSP430 core, and for the MSP430X core when the Small code model is used, functions are called with:

```
push.w PC
add.w #(func) - ($-4), PC
```

The return address is stored on the stack and uses 2 bytes of memory. It must be adjusted before returning from the function:

```
add.w #0x4, 0(SP)
ret
```

## MSP430X using the Large code model

For the MSP430X core when the Large code model is used, functions are called with:

```
pushm.a #0x1, PC
adda #(func) - ($-4-?CPU30_OFFSET), PC
```

The return address is stored on the stack and uses 4 bytes of memory. It must be adjusted before returning from the function:

```
addx.a #0x4, 0(SP)
reta
```

**Note:** The symbol `?CPU30_OFFSET` is given different values in the device-specific linker configuration files, to compensate for hardware defect CPU30.

## INTERRUPT FUNCTIONS

Interrupt functions always return using the `RETI` instructions. This restores the status registers and returns to the location where the interrupt occurred.

## MODULE CONSISTENCY

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

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:

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

Also, for MSP430X devices, define the runtime attribute `__code_model`. Use one of:

```
RTMODEL "__code_model", "small"
RTMODEL "__code_model", "large"
```

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

---

## Memory access methods

This section describes the different memory types presented in the chapter *Data storage*. In addition to presenting the assembler code used for accessing data, this section will explain the reason behind the different memory types.

You should be familiar with the MSP430 instruction set, in particular the different addressing modes used by the instructions that can access memory.

For each of the access methods described in the following sections, there are three examples:

- Accessing a global variable
- Accessing a global array using an unknown index
- Accessing a structure using a pointer.

These three examples can be illustrated by this C program:

```
char myVar;
char MyArr[10];

struct MyStruct
{
 long mA;
 char mB;
};

char Foo(int i, struct MyStruct *p)
{
 return myVar + MyArr[i] + p->mB;
}
```

## THE DATA16 MEMORY ACCESS METHOD

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.

### Examples

These examples access data16 memory in different ways:

```
mov.b &myVar, R14

mov.b MyArr(R12), R14

mov.b 0x4(R13), R14
```

## THE DATA20 MEMORY ACCESS METHOD

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.

### Examples

These examples access data20 memory in different ways:

```
movx.b &myVar, R14

movx.b MyArr (R12), R14

mov.b 0x4 (R13), R14
```

---

## Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *IAR Assembler User Guide for MSP430*.

### CFI DIRECTIVES

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

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

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

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

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

| Resource | Description                                                |
|----------|------------------------------------------------------------|
| CFA      | The call frames of the stack                               |
| R4–R15   | Normal registers                                           |
| R4L–R15L | Lower 16 bits, when compiling for the MSP430X architecture |
| R4H–R15H | Higher 4 bits, when compiling for the MSP430X architecture |
| SP       | The stack pointer                                          |
| SR       | The processor state register                               |
| PC       | The program counter                                        |

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

## CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

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

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

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

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



On the command line, use the option `-IA`.



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

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

```

NAME `cfi`

RTMODEL "__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, R4:16, R5:16, R6:16,
 R7:16, R8:16
CFI Resource R9:16, R10:16, R11:16, R12:16, 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

RSEG CODE:CODE:REORDER:NOROOT(1)
cfiExample:
 CFI Block cfiBlock0 Using cfiCommon0
 CFI Function cfiExample
 FUNCALL cfiExample, F
 LOCFRAME CSTACK, 4, STACK
 PUSH.W R10
 CFI R10 Frame(CFA, -4)
 CFI CFA SP+4
 MOV.W R12, R10
 MOV.W R10, R12
 CALL #F
 ADD.W R12, R10
 MOV.W R10, R12
 POP.W R10
 CFI R10 SameValue
 CFI CFA SP+2
 RET
 CFI 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

- C language overview
- Extensions overview
- 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 defined immediately after the keyword should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a `for` loop
- The `bool` data type
- The `long long` data type
- The `complex` floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator `_Pragma()`
- Variadic macros, which are the preprocessor macro equivalents of `printf` style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option `--vla`

- Inline assembler using the `asm` or the `__asm` keyword, see *Inline assembler*, page 169.

**Note:** Even though it is a C99 feature, the IAR C/C++ Compiler for MSP430 does not support UCNs (universal character names).

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

---

## Extensions overview

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

This is an overview of the available extensions:

- IAR C language extensions

For information about available language extensions, see *IAR C language extensions*, page 195. For more information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

The `#pragma` directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages.

Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For information about available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

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

- Intrinsic functions

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

*assembler*, page 167. For information about available functions, see the chapter *Intrinsic functions*.

- Library functions

The DLIB runtime environment provides the C and C++ library definitions in the C/C++ standard library that apply to embedded systems. For more information, see DLIB runtime environment—implementation details, page 371.

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

## ENABLING LANGUAGE EXTENSIONS

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

| Command line          | IDE*                                | Description                                                                                                                                                                                     |
|-----------------------|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--strict</code> | <b>Strict</b>                       | All IAR C language extensions are disabled; errors are issued for anything that is not part of Standard C.                                                                                      |
| None                  | <b>Standard</b>                     | All <i>extensions to Standard C</i> are enabled, but no <i>extensions for embedded systems programming</i> . For information about extensions, see <i>IAR C language extensions</i> , page 195. |
| <code>-e</code>       | <b>Standard with IAR extensions</b> | All <i>IAR C language extensions</i> are enabled.                                                                                                                                               |

Table 28: Language extensions

\* In the IDE, choose **Project>Options>C/C++ Compiler>Language>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

## IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific 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 198.

## 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 230, and *location*, page 335.

- Alignment control

Each data type has its own alignment; for more information, see *Alignment*, page 295. If you want to change the alignment, the `#pragma pack`, and the `#pragma data_alignment` directives are available. If you want to check the alignment of an object, use the `__ALIGNOF__()` operator.

The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

- `__ALIGNOF__(type)`
- `__ALIGNOF__(expression)`

In the second form, the expression is not evaluated.

- Anonymous structs and unions

C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 229.

- 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 298.

- `static_assert()`

The construction `static_assert(const-expression, "message");` can be used in C/C++. The construction will be evaluated at compile time and if `const-expression` is false, a message will be issued including the `message` string.

- Parameters in variadic macros

Variadic macros are the preprocessor macro equivalents of `printf` style functions. The preprocessor accepts variadic macros with no arguments, which means if no parameter matches the `...` parameter, the comma is then deleted in the `" , ##__VA_ARGS__"` macro definition. According to Standard C, the `...` parameter must be matched with at least one argument.

## Dedicated segment operators

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

|                              |                                                                       |
|------------------------------|-----------------------------------------------------------------------|
| <code>__segment_begin</code> | Returns the address of the first byte of the named segment.           |
| <code>__segment_end</code>   | Returns the address of the first byte <i>after</i> the named segment. |
| <code>__segment_size</code>  | Returns the size of the named segment in bytes.                       |

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

These operators behave syntactically as if declared like:

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

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

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

**Note:** These segment operators always return the linked address of a segment. When ROPI is used, you must add the code distance to get the actual address, see `__code_distance`, page 350.

### Example

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

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

See also segment, page 342, and location, page 335.

## 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 302.
- Taking the address of a register variable
 

In Standard C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.
- `long float` means `double`

The type `long float` is accepted as a synonym for `double`.
- Repeated `typedef` declarations
 

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.
- Mixing pointer types
 

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

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.

- Non-top level `const`

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-lvalue arrays

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

- Comments at the end of preprocessor directives

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

- An extra comma at the end of `enum` lists

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

- A label preceding a `}`

In Standard C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler allows this, but issues a warning.

Note that this also applies to the labels of `switch` statements.

- Empty declarations

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

- Single-value initialization

Standard C requires that all initializer expressions of static arrays, structs, and unions are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:

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

- Declarations in other scopes

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

```
int test(int x)
{
 if (x)
 {
 extern int y;
 y = 1;
 }

 return y;
}
```

- Expanding function names into strings with the function as context

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

```
"void func(char)"
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 270.

- Static functions in function and block scopes

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

- Numbers scanned according to the syntax for numbers

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



# Using C++

- Overview—EC++ and EEC++
- Enabling support for C++
- EC++ feature descriptions
- EEC++ feature description
- C++ language extensions

---

## Overview—EC++ and EEC++

IAR Systems supports the C++ language. You can choose between the industry-standard Embedded C++ and Extended Embedded C++. *Using C++* describes what you need to consider when using the C++ language.

Embedded C++ is a proper subset of the C++ programming language which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language. EC++ offers the same object-oriented benefits as C++, but without some features that can increase code size and execution time in ways that are hard to predict.

### EMBEDDED C++

These C++ features are supported:

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

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are features added very late before Standard C++ was defined. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

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

- Templates
- Multiple and virtual inheritance
- Exception handling
- Runtime type information
- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

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

## EXTENDED EMBEDDED C++

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

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

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

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++

language, which means no exceptions and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

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

---

## Enabling support for C++



In the compiler, the default language is C.

To compile files written in Embedded C++, use the `--ec++` compiler option. See `--ec++`, page 270.

To take advantage of *Extended* Embedded C++ features in your source code, use the `--eec++` compiler option. See `--eec++`, page 271.

For EC++, and EEC++, you must also use the IAR DLIB runtime library.



To enable EC++ or EEC++ in the IDE, choose **Project>Options>C/C++ Compiler>Language** and select the appropriate standard.

---

## EC++ feature descriptions

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

### USING IAR ATTRIBUTES WITH CLASSES

Static data members of C++ classes are treated the same way global variables are, and can have any applicable IAR type, memory, and object attribute.

Member functions are in general treated the same way free functions are, and can have any applicable IAR type, memory, and object attributes. Virtual member functions can only have attributes that are compatible with default function pointers, and constructors and destructors cannot have any such attributes.

The location operator `@` and the `#pragma location` directive can be used on static data members and with all member functions.

### Example of using attributes with classes

```
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();

 // A virtual task function
 virtual __task void H();

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

### The this pointer

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

### Class memory

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

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

**Example**

```

class __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 60.

## 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.

```
#include <stddef.h>

// 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,

```
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.

## USING STATIC CLASS OBJECTS IN INTERRUPTS

If interrupt functions use static class objects that need to be constructed (using constructors) or destroyed (using destructors), your application will not work properly if the interrupt occurs before the objects are constructed, or, during or after the objects are destroyed.

To avoid this, make sure that these interrupts are not enabled until the static objects have been constructed, and are disabled when returning from `main` or calling `exit`. For information about system startup, see *System startup and termination*, page 135.



Function local static class objects are constructed the first time execution passes through their declaration, and are destroyed when returning from `main` or when calling `exit`.

## USING NEW HANDLERS

To handle memory exhaustion, you can use the `set_new_handler` function.

### New handlers in Embedded C++

If you do not call `set_new_handler`, or call it with a NULL new handler, and `operator new` fails to allocate enough memory, it will call `abort`. The `nothrow` variant of the new operator will instead return NULL.

If you call `set_new_handler` with a non-NULL new handler, the provided new handler will be called by `operator new` if `operator new` fails to allocate memory. The new handler must then make more memory available and return, or abort execution in some manner. The `nothrow` variant of `operator new` will never return NULL in the presence of a new handler.

## TEMPLATES

Extended EC++ supports templates according to the C++ standard, but not the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

## DEBUG SUPPORT IN C-SPY

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

**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.

For more information about this, see the *C-SPY® Debugging Guide for MSP430*.

---

## EEC++ feature description

This section describes features that distinguish Extended EC++ from EC++.

## TEMPLATES

The compiler supports templates with the syntax and semantics as defined by Standard C++. However, note that the STL (standard template library) delivered with the product is tailored for Extended EC++, see *Extended Embedded C++*, page 202.

### 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**

```
// We assume that __data20 is the memory type of the default
// pointer.
template<typename T> void fun(T *);

void MyF()
{
 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**

```
// We assume that __data20 is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
 fun((int __data16 *) 0); // T = int __data16
}
```

**Non-type template parameters**

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

**Example**

```
#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 `--regvar_r4` compiler option.

**Example**

```
#include <vector>

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
```

**The standard template library**

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

**Example**

```
#include <vector>

vector<int> D; // D placed in default memory,
 // using the default heap,
 // uses default pointers
vector<int __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
```

Note that this is illegal:

```
vector<int __data16> __data20 Z;
```

Note also that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T> mem` where `mem` is the memory type of `T`. Supplying a key with a memory type is not useful.

**Example**

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated `assign` member method must be used.

```
#include <vector>

vector<int __data16> X;
vector<int __data20> Y;

void MyF()
{
 // The templated assign member method will work
 X.assign(Y.begin(), Y.end());
 Y.assign(X.begin(), X.end());
}
```

**VARIANTS OF CAST OPERATORS**

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

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

## MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

## NAMESPACE

The namespace feature is only supported in *Extended* EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

## THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

---

## C++ language extensions

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

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

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

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

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

According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:

```
struct A
{
 int A::F(); // Possible when using IAR language extensions
 int G(); // According to the standard
};
```

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

```
extern "C" void F(); // Function with C linkage
void (*PF)() // PF points to a function with C++ linkage
 = &F; // Implicit conversion of function pointer.
```

According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```
bool X;

char *P1 = X ? "abc" : "def"; //Possible when using IAR
 //language extensions
char const *P2 = X ? "abc" : "def"; //According to the standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.
- An anonymous union can be introduced into a containing class by a `typedef` name. It is not necessary to first declare the union. For example:

```
typedef union
{
 int i,j;
} U; // U identifies a reusable anonymous union.

class A
{
public:
 U; // OK -- references to A::i and A::j are allowed.
};
```

In addition, this extension also permits *anonymous classes* and *anonymous structs*, as long as they have no C++ features (for example, no static data members or member

functions, and no non-public members) and have no nested types other than other anonymous classes, structs, or unions. For example:

```
struct A
{
 struct
 {
 int i,j;
 }; // OK -- references to A::i and A::j are allowed.
};
```

- The friend class syntax allows nonclass types as well as class types expressed through a typedef without an elaborated type name. For example:

```
typedef struct S ST;

class C
{
public:
 friend S; // Okay (requires S to be in scope)
 friend ST; // Okay (same as "friend S;")
 // friend S const; // Error, cv-qualifiers cannot
 // appear directly
};
```

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



# Application-related considerations

- Stack considerations
- Heap considerations
- Interaction between the tools and your application
- Checksum calculation for verifying image integrity

---

## Stack considerations

To make your application use stack memory efficiently, there are some considerations to be made.

### STACK SIZE CONSIDERATIONS

The required stack size depends heavily on the application's behavior. If the given stack size is too large, RAM will be wasted. If the given stack size is too small, one of two things can happen, depending on where in memory you located your stack:

- Variable storage will be overwritten, leading to undefined behavior
- The stack will fall outside of the memory area, leading to an abnormal termination of your application.

Both alternatives are likely to result in application failure. Because the second alternative is easier to detect, you should consider placing your stack so that it grows toward the end of the RAM memory.

For more information about the stack size, see *Setting up stack memory*, page 105, and *Saving stack space and RAM memory*, page 240.

---

## Heap considerations

The heap contains dynamic data allocated by use of the C function `malloc` (or a corresponding function) 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, see *Setting up heap memory*, page 106.

## 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
```

If you use any of the standard functions without a prefix, the function will be mapped to the default memory type `data16`.

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

For information about available heaps, see *Setting up heap memory*, page 106.

## HEAP SEGMENTS IN CLIB

The memory allocated to the heap is placed in the segment `HEAP`, which is only included in the application if dynamic memory allocation is actually used.

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 AND STANDARD I/O



If you excluded `FILE` descriptors from the DLIB runtime environment, as in the Normal configuration, there are no input and output buffers at all. Otherwise, as in the Full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an MSP430 microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

## Interaction between the tools and your application

The linking process and the application can interact symbolically in four ways:

- Creating a symbol by using the linker command line option `-D`. The linker will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.
- Using the compiler operators `__segment_begin`, `__segment_end`, or `__segment_size`, or the assembler operators `SFB`, `SFE`, or `SIZEOF` on a named segment. These operators provide access to the start address, end address, and size of a contiguous sequence of segments with the same name
- The command line option `-s` informs the linker about the start label of the application. It is used by the linker as a root symbol and to inform the debugger where to start execution.

The following lines illustrate how to use `-D` to create a symbol. If you need to use this mechanism, add these options to your command line like this:

```
-Dmy_symbol=A
-DMY_HEAP_SIZE=400
```

The linker configuration file can look like this:

```
-Z (DATA) MyHeap+MY_HEAP_SIZE=20000-2FFFF
```

Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by an XLINK option to dynamically allocate
an array of elements with specified size. The value takes the
form of a label.
*/
extern int NrOfElements;

typedef char Elements;
Elements *GetElementArray()
{
 return malloc(sizeof(Elements) * (long) &NrOfElements);
}

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

```

/* Declare the section that contains the heap. */
#pragma segment = "MYHEAP"

char *MyHeap()
{
 /* First get start of statically allocated section, */
 char *p = __segment_begin("MYHEAP");

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

```

---

## Checksum calculation for verifying image integrity

This section contains information about checksum calculation:

- *Briefly about checksum calculation*, page 220
- *Calculating and verifying a checksum*, page 222
- *Troubleshooting checksum calculation*, page 226

### BRIEFLY ABOUT CHECKSUM CALCULATION

You can use a checksum to verify that the image is the same at runtime as when the image's original checksum was generated. In other words, to verify that the image has not been corrupted.

This works as follows:

- You need an initial checksum.  
You can either use the linker to generate an initial checksum or you might have a third-party checksum available.
- You must generate a second checksum during runtime.  
You can either add specific code to your application source code for calculating a checksum during runtime or you can use some dedicated hardware on your device for calculating a checksum during runtime.
- You must add specific code to your application source code for comparing the two checksums and take an appropriate action if they differ.

If the two checksums have been calculated in the same way, and if there are no errors in the image, the checksums should be identical. If not, you should first suspect that the two checksums were not generated in the same way.

No matter which solutions you use for generating the two checksum, you must make sure that both checksums are calculated *in the exact same way*. If you use the linker for the initial checksum and use a software-based calculation during runtime, you have full control of the generation for both checksums. However, if you are using a third-party checksum for the initial checksum or some hardware support for the checksum calculation during runtime, there might be additional requirements that you must consider.

For the two checksums, there are some choices that you must always consider and there are some choices to make only if there are additional requirements. Still, all of the details must be the same for both checksums.

Details always to consider:

- *Checksum range*

The memory range (or ranges) that you want to verify by means of checksums.

Typically, you might want to calculate a checksum for all ROM memory. However, you might want to calculate a checksum only for specific ranges. Remember that:

- It is OK to have several ranges for one checksum.
- Typically, the checksum must be calculated from the lowest to the highest address for every memory range.
- Each memory range must be verified in the same order as defined (for example, 0x100-0x1FF,0x400-0x4FF is not the same as 0x400-0x4FF,0x100-0x1FF).
- If several checksums are used, you should place them in sections with unique names and use unique symbol names.
- A checksum should never be calculated on a memory range that contains a checksum or a software breakpoint.

- *Algorithm and size of checksum*

You should consider which algorithm is most suitable in your case. There are two basic choices, Sum (a simple arithmetic algorithm) or CRC (which is the most commonly used algorithm). For CRC there are different sizes to choose for the checksum, 2 or 4 bytes where the predefined polynomials are wide enough to suit the size, for more error detecting power. The predefined polynomials work well for most, but possibly not for all data sets. If not, you can specify your own polynomial. If you just want a decent error detecting mechanism, use the predefined CRC algorithm for your checksum size, typically CRC16 or CRC32.

Note that for an  $n$ -bit polynomial, the  $n$ :th bit is always considered to be set. For a 16-bit polynomial (for example, CRC16) this means that 0x11021 is the same as 0x1021.

For more information about selecting an appropriate polynomial for data sets with non-uniform distribution, see for example section 3.5.3 in *Tannenbaum, A.S., "Computer Networks," Prentice Hall 1981, ISBN: 0131646990*.

- *Fill*

Every byte in the checksum range must have a well-defined value before the checksum can be calculated. Typically, bytes with unknown values are *pad bytes* that have been added for alignment. This means that you must specify which fill pattern to be used during calculation, typically `0xFF` or `0x00`.

- *Initial value*

The checksum must always have an explicit initial value.

In addition to these mandatory details, there might be other details to consider. Typically, this might happen when you have a third-party checksum, you want the checksum be compliant with the Rocksoft™ checksum model, or when you use hardware support for generating a checksum during runtime. The linker provides support for also controlling alignment, complement, bit order, and checksum unit size.

## CALCULATING AND VERIFYING A CHECKSUM

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

- 1 The `CHECKSUM` segment will only be included in your application if the segment appears to be needed. If the checksum is not needed by the application itself, use the linker option `-g__checksum` to force the segment to be included.
- 2 When configuring the linker to calculate a checksum, there are some basic choices to make:
  - Checksum algorithm  
Choose which checksum algorithm you want to use. In this example, the CRC16 algorithm is used.
  - Memory range  
Using the IDE, the checksum will by default be calculated for all placement directives (specified in the linker configuration file) for ROM-based memory. From the command line, you can specify any ranges.
  - Fill pattern  
Specify a fill pattern—typically `0xFF` or `0x00`—for bytes with unknown values. The fill pattern will be used in all checksum ranges.

For more information, see *Briefly about checksum calculation*, page 220.



To run the linker from the IDE, choose **Project>Options>Linker>Checksum** and make your settings, for example:

In the simplest case, you can ignore (or leave with default settings) these options: **Complement**, **Bit order**, and **Checksum unit size**.



To make the linker create a checksum from the command line, use the `-J` linker option, for example like this:

```
-J2, crc16, , __checksum, CHECKSUM, 1=0x8002-0x8FFF
```

The checksum will be created when you build your project and will be placed in the automatically generated segment `CHECKSUM`. If you are using your own linker configuration file or if you explicitly want to control the placement of the `CHECKSUM` segment, you must update your linker configuration file with placement information accordingly. In that case, make sure to place the segment so that it is not part of the application's checksum calculation.

**3** You can specify several ranges instead of only one range.



If you are using the IDE, perform these steps:

- Choose **Project>Options>Linker>Checksum** and make sure to deselect **Fill unused code memory**.
- Choose **Project>Options>Linker>Extra Options** and specify the ranges, for example like this:

```
-h (CODE) 0-3FF, 8002-8FFF
-J2, crc16, , 1=0-3FF, 8002-8FFF
```



If you are using the command line, use the `-J` option and specify the ranges. for example like this:

```
-h(CODE) 0-3FF, 8002-8FFF
-J2, crc16, ,, 1=0-3FF, 8002-8FFF
```

- 4 Add a function for checksum calculation to your source code. Make sure that the function uses the same algorithm and settings as for the checksum calculated by the linker. For example, a slow variant of the `crc16` algorithm but with small memory footprint (in contrast to the fast variant that uses more memory):

```
unsigned short SmallCrc16(uint16_t
 sum,
 unsigned char *p,
 unsigned int len)
{
 while (len--)
 {
 int i;
 unsigned char byte = *(p++);

 for (i = 0; i < 8; ++i)
 {
 unsigned long oSum = sum;
 sum <<= 1;
 if (byte & 0x80)
 sum |= 1;
 if (oSum & 0x8000)
 sum ^= 0x1021;
 byte <<= 1;
 }
 }
 return sum;
}
```

You can find the source code for this checksum algorithm in the `430\src\linker` directory of your product installation.



- 5** Make sure that your application also contains a call to the function that calculates the checksum, compares the two checksums, and takes appropriate action if the checksum values do not match. This code gives an example of how the checksum can be calculated for your application and to be compared with the linker generated checksum:

```

/* The calculated checksum */

/* Linker generated symbols */
extern unsigned short const __checksum;
extern int __checksum_begin;
extern int __checksum_end;

void TestChecksum()
{
 unsigned short calc = 0;
 unsigned char zeros[2] = {0, 0};

 /* Run the checksum algorithm */
 calc = SmallCrc16(0,
 (unsigned char *) &__checksum_begin,
 ((unsigned char *) &__checksum_end -
 ((unsigned char *) &__checksum_begin)+1));

 /* Fill the end of the byte sequence with zeros. */
 calc = SmallCrc16(calc, zeros, 2);

 /* Test the checksum */
 if (calc != __checksum)
 {
 printf("Incorrect checksum!\n");
 abort(); /* Failure */
 }

 /* Checksum is correct */
}

```

**Note:** Make sure to define the symbols `__checksum_begin` and `__checksum_end` in your linker configuration file.

- 6** Build your application project and download it.

During the build, the linker creates a checksum and places it in the specified symbol `__checksum` in the segment `CHECKSUM`.

- 7** Choose **Download and Debug** to start the C-SPY debugger.

During execution, the checksum calculated by the linker and the checksum calculated by your application should be identical.

## TROUBLESHOOTING CHECKSUM CALCULATION

If the two checksums do not match, there are several possible causes. These are some troubleshooting hints:

- If possible, start with a small example when trying to get the checksums to match.
- Verify that the exact same memory range or ranges are used in both checksum calculations.
 

To help you do this, The linker produces useful information in the map file about the exact addresses that were used and the order in which they were accessed.
- Make sure that all checksum symbols are excluded from all checksum calculations.
 

In the map file, notice the checksum symbol and its size, and for information about its placement, check the module map or the entry list. Compare that placement with the checksum range.
- Verify that the checksum calculations use the same polynomial.
- Verify that the bits in the bytes are processed in the same order in both checksum calculations, from the least to the most significant bit or the other way around. You control this with the **Bit order** option (or from the command line, the `-m` parameter of the `--checksum` option).
- If you are using the small variant of CRC, check whether you need to feed additional bytes into the algorithm.
 

The number of zeros to add at the end of the byte sequence must match the size of the checksum, in other words, one zero for a 1-byte checksum, two zeros for a 2-byte checksum, and four zeros for a 4-byte checksum.
- Any breakpoints in flash memory change the content of the flash. This means that the checksum which is calculated by your application will no longer match the initial checksum calculated by the linker. To make the two checksums match again, you must disable all your breakpoints in flash and any breakpoints set in flash by C-SPY internally. The stack plugin and the debugger option **Run to** both require C-SPY to set breakpoints. Read more about possible breakpoint consumers in the *C-SPY® Debugging Guide for MSP430*.

# Efficient coding for embedded applications

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

---

## 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.

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

### FLOATING-POINT TYPES

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

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

In the compiler, the floating-point type `float` always uses the 32-bit format. 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 because 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`:

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

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

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

For more information about floating-point types, see Basic data types—floating-point types, page 299.

## 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 situations when this can be a problem:

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

For information about alignment requirements, see Alignment, page 295.

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.

Alternatively, write your own customized functions for *packing* and *unpacking* structures. This is a more portable way, which will not produce any more code apart from your functions. The drawback is the need for two views on the structure data—packed and unpacked.

For more information about the `#pragma pack` directive, see `pack`, page 339.

## 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 270, for additional information.

### Example

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

```
struct S
{
 char mTag;
 union
 {
 long mL;
 float mF;
 };
} St;

void F(void)
{
 St.mL = 5;
}
```

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

```
__no_init volatile
union
{
 unsigned char IOPORT;
 struct
 {
 unsigned char way: 1;
 unsigned char out: 1;
 };
} @ 8;

/* The variables are used here. */
void Test(void)
{
 IOPORT = 0;
 way = 1;
 out = 1;
}
```

This declares an I/O register byte `IOPORT` at address 8. 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 and know which one is best suited for different situations. You can use:

- Data models (MSP430X only)
  - By selecting a data model, you can control the default memory placement of variables and constants. For more information, see *Data models*, page 66.

- Memory attributes (MSP430X only)  
Using IAR-specific keywords or pragma directives, you can override the default placement of variables and constants. For more information, see *Using data memory attributes*, page 61.
- The @ operator and the #pragma location directive for absolute placement.  
Using the @ operator or the #pragma location directive, you can place individual global and static variables at absolute addresses. For more information, see *Data placement at an absolute location*, page 231.
- The @ operator and the #pragma location directive for segment placement.  
Using the @ operator or the #pragma location directive, you can place individual functions, variables, and constants in named segments. The placement of these segments can then be controlled by linker directives. For more information, see *Data and function placement in segments*, page 233
- Using the --segment option, you can set the default segment for functions, variables, and constants in a particular module. For more information, see --segment, page 289.

## DATA PLACEMENT AT AN ABSOLUTE LOCATION

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

- \_\_no\_init
- \_\_no\_init and const (without initializers)
- const (with initializers)

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

**Note:** All declarations of variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

## Examples

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

```
__no_init volatile char alpha @ 0x200; /* OK */
```

The next example contains two `const` declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=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.

This shows incorrect usage:

```
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:

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

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

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

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

**Note:** C++ static member variables can be placed at an absolute address just like any other static variable.



## DATA AND FUNCTION PLACEMENT IN SEGMENTS

The following 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 `--segment` option can be used for placing variables and functions, which are parts of the whole compilation unit, in named segments.

C++ static member variables can be placed in named segments just like any other static variable.

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker configuration file using the `-Z` or the `-P` segment control directives.

**Note:** Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

The location of the segments can be controlled from the linker configuration file.

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

### Examples of placing variables in named segments

In the following examples, a data object is placed in a user-defined segment. If no memory attribute is specified, the variable will, like any other variable, be treated as if it is located in the default memory. Note that you must as always ensure that the segment is placed in the appropriate memory area when linking.

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

#pragma location="MY_CONSTANTS"
const int beta = 42; /* OK */

const int gamma @ "MY_CONSTANTS" = 17; /* OK */
int theta @ "MY_ZEROS"; /* OK */
int phi @ "MY_INITED" = 4711; /* OK */
```

The compiler will warn that segments that contain zero-initialized and initialized data must be handled manually. To do this, you must use the linker option `-Q` to separate the initializers into one separate segment and the symbols to be initialized to a different

segment. You must then write source code that copies the initializer segment to the initialized segment, and zero-initialized symbols must be cleared before they are used.

As described elsewhere, for MSP430X you can use memory attributes to select a memory for the variable. Note that you must as always ensure that the segment is placed in the appropriate memory area when linking.

```
__data16 __no_init int alpha @ "MY_DATA16_NOINIT"; /* Placed in
 data16*/
```

### Examples of placing functions in named segments

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

void g(void) @ "MY_FUNCTIONS"
{
}

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

---

## Controlling compiler optimizations

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

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

### SCOPE FOR PERFORMED OPTIMIZATIONS

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

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. See `optimize`, page 337, 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 276.

**Note:** Only one object file is generated, and thus all symbols will be part of that object file.

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 268.

## OPTIMIZATION LEVELS

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

| Optimization level        | Description                                                                                                                                                   |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| None (Best debug support) | Variables live through their entire scope<br>Dead code elimination<br>Redundant label elimination<br>Redundant branch elimination                             |
| Low                       | Same as above but variables only live for as long as they are needed, not necessarily through their entire scope                                              |
| Medium                    | Same as above, and:<br>Live-dead analysis and optimization<br>Code hoisting<br>Register content analysis and optimization<br>Common subexpression elimination |
| High (Balanced)           | Same as above, and:<br>Peephole optimization<br>Cross jumping<br>Loop unrolling<br>Function inlining<br>Type-based alias analysis                             |

Table 29: 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 237.

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

## SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes

enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

If you use the optimization level High speed, the `--no_size_constraints` compiler option relaxes the normal restrictions for code size expansion and enables more aggressive optimizations.

You can choose an optimization goal for each module, or even individual functions, using command line options and pragma directives (see `-O`, page 283 and `optimize`, page 337). For a small embedded application, this makes it possible to achieve acceptable speed performance while minimizing the code size: Typically, only a few places in the application need to be fast, such as the most frequently executed inner loops, or the interrupt handlers.

Rather than compiling the whole application with High (Balanced) optimization, you can use High (Size) in general, but override this to get High (Speed) optimization only for those functions where the application needs to be fast.

Because of the unpredictable way in which different optimizations interact, where one optimization can enable other optimizations, sometimes a function becomes smaller when compiled with High (Speed) optimization than if High (Size) is used. Also, using multi-file compilation (see `--mfc`, page 276) can enable many optimizations to improve both speed and size performance. It is recommended that you experiment with different optimization settings so that you can pick the best ones for your project.

## 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.

For more information about the command line option, see `--no_cse`, page 278.

### Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most efficient for smaller loops, where the loop overhead can be a substantial part of the total loop body.

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

The compiler heuristically decides which loops to unroll. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

To disable loop unrolling, use the command line option `--no_unroll`, see `--no_unroll`, page 282.

### Function inlining

Function inlining means that a function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization normally reduces execution time, but might increase the code size.

For more information, see *Inlining functions*, page 81.

### 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 Medium and above, 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 below Medium.

For more information about the command line option, see `--no_code_motion`, page 278.

### Type-based alias analysis

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

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

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

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

For more information about the command line option, see `--no_tbaa`, page 281.

### Example

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

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

---

## Facilitating good code generation

This section contains hints on how to help the compiler generate good code:

- *Writing optimization-friendly source code*, page 240
- *Saving stack space and RAM memory*, page 240
- *Function prototypes*, page 240
- *Integer types and bit negation*, page 241
- *Protecting simultaneously accessed variables*, page 242
- *Accessing special function registers*, page 243
- *Non-initialized variables*, page 245
- *Efficient switch statements*, page 245

## WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables (non-static). Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 238. 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 235.
- Avoid using inline assembler. Instead, try writing the code in C/C++, use intrinsic functions, or write a separate module in assembler language. For more information, see *Mixing C and assembler*, page 167.

## SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that 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 `--reduce_stack_usage` option. This will eliminate holes in the stack but results in somewhat larger code. See `--reduce_stack_usage`, page 286.

## FUNCTION PROTOTYPES

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

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



Both styles are valid C, however it is strongly recommended to use the prototyped style, and provide a prototype declaration for each public function in a header that is included both in the compilation unit defining the function and in all compilation units using it.

The compiler will not perform type checking on parameters passed to functions declared using K&R style. Using prototype declarations will also result in more efficient code in some cases, as there is no need for type promotion for these functions.

To make the compiler require that all function definitions use the prototyped style, and that all public functions have been declared before being defined, use the **Project>Options>C/C++ Compiler>Language 1>Require prototypes** compiler option (`--require_prototypes`).

### Prototyped style

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

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

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

### Kernighan & Ritchie style

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

For example:

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

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

## INTEGER TYPES AND BIT NEGATION

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

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 16-bit integer, and two's complement is assumed:

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

Here, the test is always false. On the right hand side, `0x80` is `0x0080`, and `~0x0080` becomes `0xFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 8 bits set.

## PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code.

For sequences of accesses to variables that you do not want to be interrupted, use the `__monitor` keyword. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. Accessing a small-sized `volatile` variable can be an atomic operation, but you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation. For more information, see `__monitor`, page 315.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see `Declaring objects volatile`, page 305.



### Protecting the eeprom write mechanism

A typical example of when it can be necessary to use the `__monitor` keyword is when protecting the eeprom write mechanism, which can be used from two threads (for example, main code and interrupts). Servicing an interrupt during an EEPROM write sequence can in many cases corrupt the written data.

## ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for all MSP430 devices are included in the IAR product installation. There are two types of header files, named `iodevice.h` and `mSPdevice.h`, respectively. They define the processor-specific special function registers (SFRs).

You can also use the header files as templates when you create new header files for other MSP430 devices.

### **`iodevice.h` files**

The header files named `iodevice.h` support C and C++. They define SFRs using structs and unions, and provide access to individual bits through bitfields.

The following example is from `io430x14x.h`:

```

/* 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 WDCNTCL : 1;
 unsigned short WDTTMSEL : 1;
 unsigned short WDTNMI : 1;
 unsigned short WDTNMIES : 1;
 unsigned short WDTMIES : 1;
 unsigned short WDTMIES : 1;
 unsigned short WDTMIES : 1;
 unsigned short WDTMIES : 1;
 unsigned short : 8;
 } WDTCTL_bit;
} @ 0x0120;

enum {
 WDTIS0 = 0x0001,
 WDTIS1 = 0x0002,
 WDTSSSEL = 0x0004,
 WDCNTCL = 0x0008,
 WDTTMSEL = 0x0010,
 WDTNMI = 0x0020,
 WDTNMIES = 0x0040,
 WDTMIES = 0x0080
};

#define WDTPW (0x5A00)

```

By including the appropriate header 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:

```
/* Object access */
WDCTL = 0x1234;

/* Bitfield accesses */
WDCTL_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:

```
WDCTL = WDTPW | WDTTHOLD; /* Stop watchdog timer */
```

For information about the @ operator, see *Placing located data*, page 104.

### mspdevice.h files

The header files named `mspdevice.h` support C, C++, and assembler. They define SFRs using primitive types, and provide access to individual bits through preprocessor defines.

The following example is from `msp430x14x.h`:

```
/* Watchdog Timer Control */
__no_init volatile unsigned short WDCTL @ 0x0120u;

/* The bit names have been prefixed with "WDT" */
#define WDTIS0 (0x0001u)
#define WDTIS1 (0x0002u)
#define WDTSSEL (0x0004u)
#define WDTCNTCL (0x0008u)
#define WDTTMSSEL (0x0010u)
#define WDTNMI (0x0020u)
#define WDTNMIES (0x0040u)
#define WDTTHOLD (0x0080u)

#define WDTPW (0x5A00u)
```

By including the appropriate header file in your source code, you can access the object as follows:

```
/* Object access */
WDTCTL = 0x1234;

/* Bitfield accesses */
WDTCTL |= WDTSSSEL;

/* Multiple bits access */
WDTCTL = (WDTPW | WDTWTHOLD); /* Stop watchdog timer */
```

## 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 *Linking overview* for more information.

For `__no_init`, the `const` keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

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

For more information, see `__no_init`, page 317. Note that to use this keyword, language extensions must be enabled; see `-e`, page 270. For more information, see also `object_attribute`, page 337.

## EFFICIENT SWITCH STATEMENTS

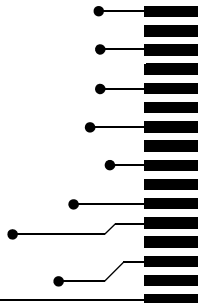
The compiler provides a way to generate very efficient code for switch statements when it is known that the value in the expression is even and within a specific limit. This can for example be used for writing efficient interrupt service routines that use the Interrupt Vector Generators Timer A, Timer B, the I<sup>2</sup>C module, and the ADC12 module. For more information, see *Interrupt Vector Generator interrupt functions*, page 74.



# Part 2. Reference information

This part of the *IAR C/C++ Compiler User Guide for MSP430* contains these chapters:

- External interface details
- Compiler options
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- C/C++ standard library functions
- Segment reference
- The stack usage control file
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89.







# External interface details

- Invocation syntax
- Include file search procedure
- Compiler output
- Diagnostics

---

## Invocation syntax

You can use the compiler either from the IDE or from the command line. See the *IDE Project Management and Building Guide* for information about using the compiler from the IDE.

### COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
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 as they are specified on the command line.

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

## PASSING OPTIONS

There are three different ways of passing options to the compiler:

- Directly from the command line  
Specify the options on the command line after the `icc430` command, either before or after the source filename; see *Invocation syntax*, page 249.
- Via environment variables  
The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 250.
- Via a text file, using the `-f` option; see `-f`, page 272.

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

## ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

| Environment variable   | Description                                                                                                                                                         |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>C_INCLUDE</code> | Specifies directories to search for include files; for example:<br><code>C_INCLUDE=c:\program files\iar systems\embedded<br/>workbench .n\430\inc;c:\headers</code> |
| <code>QCC430</code>    | Specifies command line options; for example: <code>QCC430=-lA<br/>asm.lst</code>                                                                                    |

Table 30: Compiler environment variables

## Include file search procedure

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

- If the name of the `#include` file is an absolute path specified in angle brackets or double quotes, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:  
`#include <stdio.h>`  
it searches these directories for the file to include:
  - 1 The directories specified with the `-I` option, in the order that they were specified, see `-I`, page 274.
  - 2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 250.
  - 3 The automatically set up library system include directories. See `--clib`, page 262, `--dlib`, page 268, and `--dlib_config`, page 269.

- 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:

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

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

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

For more information, see Overview of the preprocessor, page 361.

---

## 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 274. By default, these files will have the filename extension `lst`.
- **Optional preprocessor output files**  
A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `i`.
- **Diagnostic messages**  
Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 253.
- **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 252.
- **Size information**  
Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as *shared*.  
Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

## ERROR RETURN CODES

The compiler returns status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

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

Table 31: Error return codes

## Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

### MESSAGE FORMAT

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

```
filename, linenumber level[tag]: message
```

with these elements:

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

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

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

### SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

#### Remark

A diagnostic message that is produced when the compiler finds a source code 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 288.

#### Warning

A diagnostic message that is produced when the compiler finds a potential programming error or omission which is of concern, but which does not prevent completion of the compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see `--no_warnings`, page 282.

## 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 the chapter *Compiler options*, for information about the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for information about the pragma directives that are available for setting severity levels.

## INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler. It is produced using this form:

Internal error: *message*

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Include enough information to reproduce the problem, typically:

- The product name
- The version number of the compiler, which can be seen in the header of the list files generated by the compiler
- Your license number
- The exact internal error message text
- The source file of the application that generated the internal error
- A list of the options that were used when the internal error occurred.

# Compiler options

- Options syntax
- Summary of compiler options
- Descriptions of compiler options

---

## Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IDE.



See the online help system for information about the compiler options available in the IDE and how to set them.

### TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it can have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 250.

### RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

#### Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac2004=n
```

### Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

```
-I..\src or -I ..\src\
```

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

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

or

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

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

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

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

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

```
-lA MyList.lst
```

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

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

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

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

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

```
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 `-o`, in which case that name is used. For example:

```
icc430 prog.c -l ..\listings\
```

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

- The *current directory* is specified with a period (`.`). For example:
- ```
icc430 prog.c -l .
```
- `/` can be used instead of `\` as the directory delimiter.
 - By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

```
icc430 prog.c -l -
```

Additional rules

These rules also apply:

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

```
icc430 prog.c -l ---r
```

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

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

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

```
--diag_warning=Be0001
```

```
--diag_warning=Be0002
```

Summary of compiler options

This table summarizes the compiler command line options:

Command line option	Description
<code>--c89</code>	Specifies the C89 dialect
<code>--char_is_signed</code>	Treats <code>char</code> as signed
<code>--char_is_unsigned</code>	Treats <code>char</code> as unsigned
<code>--clib</code>	Uses the system include files for the CLIB library
<code>--code_model</code>	Specifies the code model

Table 32: Compiler options summary

Command line option	Description
--core	Specifies a CPU core
-D	Defines preprocessor symbols
--data_model	Specifies the data model
--debug	Generates debug information
--dependencies	Lists file dependencies
--diag_error	Treats these as errors
--diag_remark	Treats these as remarks
--diag_suppress	Suppresses these diagnostics
--diag_warning	Treats these as warnings
--diagnostics_tables	Lists all diagnostic messages
--discard_unused_publics	Discards unused public symbols
--dlib	Uses the system include files for the DLIB library
--dlib_config	Uses the system include files for the DLIB library and determines which configuration of the library to use
--double	Forces the compiler to use 32-bit or 64-bit doubles
-e	Enables language extensions
--ec++	Specifies Embedded C++
--eec++	Specifies Extended Embedded C++
--enable_multibytes	Enables support for multibyte characters in source files
--enable_restrict	Enables the Standard C keyword <code>restrict</code>
--error_limit	Specifies the allowed number of errors before compilation stops
-f	Extends the command line
--guard_calls	Enables guards for function static variable initialization
--header_context	Lists all referred source files and header files
--hw_workaround	Enables workarounds for various hardware issues
-I	Specifies include file path
-l	Creates a list file
--library_module	Creates a library module

Table 32: Compiler options summary (Continued)

Command line option	Description
<code>--lock_r4</code>	Excludes register R4 from use
<code>--lock_r5</code>	Excludes register R5 from use
<code>--macro_positions_in_diagnostics</code>	Obtains positions inside macros in diagnostic messages
<code>--mfc</code>	Enables multi-file compilation
<code>--migration_preprocessor_extensions</code>	Extends the preprocessor
<code>--misrac</code>	Enables error messages specific to MISRA-C:1998. This option is a synonym of <code>--misrac1998</code> and is only available for backwards compatibility.
<code>--misrac1998</code>	Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .
<code>--misrac2004</code>	Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .
<code>--misrac_verbose</code>	<i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> or the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .
<code>--module_name</code>	Sets the object module name
<code>--multiplier</code>	Enables support for the hardware multiplier
<code>--no_call_frame_info</code>	Disables output of call frame information
<code>--no_code_motion</code>	Disables code motion optimization
<code>--no_cse</code>	Disables common subexpression elimination
<code>--no_inline</code>	Disables function inlining
<code>--no_path_in_file_macros</code>	Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>
<code>--no_rw_dynamic_init</code>	Issues an error message instead of rewriting initializations.
<code>--no_size_constraints</code>	Relaxes the normal restrictions for code size expansion when optimizing for speed.
<code>--no_static_destruction</code>	Disables destruction of C++ static variables at program exit
<code>--no_system_include</code>	Disables the automatic search for system include files

Table 32: Compiler options summary (Continued)

Command line option	Description
<code>--no_tbaa</code>	Disables type-based alias analysis
<code>--no_typedefs_in_diagnostics</code>	Disables the use of typedef names in diagnostics
<code>--no_ubrof_messages</code>	Excludes messages from UBROF files
<code>--no_unroll</code>	Disables loop unrolling
<code>--no_warnings</code>	Disables all warnings
<code>--no_wrap_diagnostics</code>	Disables wrapping of diagnostic messages
<code>-O</code>	Sets the optimization level
<code>-o</code>	Sets the object filename. Alias for <code>--output</code> .
<code>--omit_types</code>	Excludes type information
<code>--only_stdout</code>	Uses standard output only
<code>--output</code>	Sets the object filename
<code>--pending_instantiations</code>	Sets the maximum number of instantiations of a given C++ template.
<code>--predef_macros</code>	Lists the predefined symbols.
<code>--preinclude</code>	Includes an include file before reading the source file
<code>--preprocess</code>	Generates preprocessor output
<code>--public_equ</code>	Defines a global named assembler label
<code>-r</code>	Generates debug information. Alias for <code>--debug</code> .
<code>--reduce_stack_usage</code>	Reduces stack usage
<code>--regvar_r4</code>	Reserves register R4 for use by global register variables
<code>--regvar_r5</code>	Reserves register R5 for use by global register variables
<code>--relaxed_fp</code>	Relaxes the rules for optimizing floating-point expressions
<code>--remarks</code>	Enables remarks
<code>--require_prototypes</code>	Verifies that functions are declared before they are defined
<code>--ropi</code>	Generates position-independent code and read-only data.
<code>--save_reg20</code>	Declares all interrupt functions <code>__save_reg20</code> by default
<code>--segment</code>	Changes a segment name

Table 32: Compiler options summary (Continued)

Command line option	Description
<code>--silent</code>	Sets silent operation
<code>--strict</code>	Checks for strict compliance with Standard C/C++
<code>--system_include_dir</code>	Specifies the path for system include files
<code>--use_cplusplus_inline</code>	Uses C++ inline semantics in C99
<code>--version</code>	
<code>--vla</code>	Enables C99 VLA support
<code>--warn_about_c_style_casts</code>	Makes the compiler warn when C-style casts are used in C++ source code
<code>--warnings_affect_exit_code</code>	Warnings affect exit code
<code>--warnings_are_errors</code>	Warnings are treated as errors

Table 32: Compiler options summary (Continued)

Descriptions of compiler options

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



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

--c89

Syntax

`--c89`

Description

Use this option to enable the C89 C dialect instead of Standard C.

Note: This option is mandatory when the MISRA C checking is enabled.

See also

C language overview, page 193.



Project>Options>C/C++ Compiler>Language 1>C dialect>C89

--char_is_signed

Syntax `--char_is_signed`

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

Note: The runtime library is compiled without the `--char_is_signed` option and cannot be used with code that is compiled with this option.



Project>Options>C/C++ Compiler>Language 2>Plain ‘char’ is

--char_is_unsigned

Syntax `--char_is_unsigned`

Description Use this option to make the compiler interpret the plain `char` type as unsigned. This is the default interpretation of the plain `char` type.



Project>Options>C/C++ Compiler>Language 2>Plain ‘char’ is

--clib

Syntax `--clib`

Description Use this option to use the system header files for the CLIB library; the compiler will automatically locate the files and use them when compiling.

See also `--dlib`, page 268 and `--dlib_config`, page 269.



To set related options, choose:

Project>Options>General Options>Library Configuration

--code_model

Syntax `--code_model={small|large}`

Parameters `small` Uses the Small code model

	large (default)	Uses the Large code model
Description	Use this option to select the code model, which controls which instruction sequences are used for function calls and returns. If you do not select a code model, the compiler uses the default code model. Note that all modules of your application must use the same code model.	
	Note: This option is only available if <code>--core</code> is set to 430X.	
See also	<i>Code models (MSP430X only)</i> , page 71	



Project>Options>General Options>Target>Code model

--core

Syntax	<code>--core={430 430X}</code>	
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 processor core for which the code will be generated. If you do not use the option to specify a core, Note that all modules of your application must use the same core.	



Project>Options>General Options>Target>Device

-D

Syntax	<code>-D symbol[=value]</code>	
Parameters	<i>symbol</i>	The name of the preprocessor symbol
	<i>value</i>	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

<code>small</code> (default)	Uses the Small data model.
<code>medium</code>	Uses the Medium data model.
<code>large</code>	Uses the Large data model.

Description

Use this option to select the data model, which controls the default placement of data objects. If you do not select a data model, the compiler uses the default data model. Note that all modules of your application must use the same data model.

Note: This option is only available if `--core` is set to 430X.

See also

Data models, page 66 and `__DATA_MODEL__`, page 363.



Project>Options>General Options>Target>Data model

--debug, -r

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

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



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

--dependencies

Syntax	<code>--dependencies=[i m n][s] {filename directory +}</code>	
Parameters	<code>i</code> (default)	Lists only the names of files
	<code>m</code>	Lists in makefile style (multiple rules)
	<code>n</code>	Lists in makefile style (one rule)
	<code>s</code>	Suppresses system files
	<code>+</code>	Gives the same output as <code>-o</code> , but with the filename extension <code>d</code>

See also *Rules for specifying a filename or directory as parameters*, page 256.

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 <code>i</code> .
-------------	--

Example	If <code>--dependencies</code> or <code>--dependencies=i</code> 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
```


Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.



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

--diag_suppress

Syntax

```
--diag_suppress=tag[, tag, ...]
```

Parameters

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

Description

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



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

--diag_warning

Syntax

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

Parameters

tag The number of a diagnostic message, for example the message number Pe826


Description

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




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

--diagnostics_tables

Syntax	<code>--diagnostics_tables {filename directory}</code>
Parameters	See <i>Rules for specifying a filename or directory as parameters</i> , page 256.
Description	<p>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.</p> <p>This option cannot be given together with other options.</p> <p> This option is not available in the IDE.</p>

--discard_unused_publics

Syntax	<code>--discard_unused_publics</code>
Description	<p>Use this option to discard unused public functions and variables when compiling with the <code>--mfc</code> compiler option.</p> <p>Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output. Use the object attribute <code>__root</code> to keep symbols that are used from outside the compilation unit, for example interrupt handlers. If the symbol does not have the <code>__root</code> attribute and is defined in the library, the library definition will be used instead.</p>
See also	<code>--mfc</code> , page 276 and <i>Multi-file compilation units</i> , page 235.
	<p> Project>Options>C/C++ Compiler>Discard unused publics</p>

--dlib

Syntax	<code>--dlib</code>
Description	<p>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.</p> <p>Note: The DLIB library is used by default: To use the CLIB library, use the <code>--clib</code> option instead.</p>

See also

`--dlib_config`, page 269, `--no_system_include`, page 280, `--system_include_dir`, page 291, and `--clib`, page 262.



To set related options, choose:

Project>Options>General Options>Library Configuration

--dlib_config

Syntax

```
--dlib_config filename.h|config
```

Parameters

<i>filename</i>	A DLIB configuration header file, see <i>Rules for specifying a filename or directory as parameters</i> , page 256.
<i>config</i>	The default configuration file for the specified configuration will be used. Choose between: <ul style="list-style-type: none"> <code>none</code>, no configuration will be used <code>normal</code>, the normal library configuration will be used <code>full</code>, the full library configuration will be used.

Description

Use this option to specify which library configuration to use, either by specifying an explicit file or by specifying a library configuration—in which case the default file for that library configuration will be used. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `430\lib`. For examples and information about prebuilt runtime libraries, see *Prebuilt runtime libraries*, page 127.

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 *Customizing and building your own runtime library*, page 124.


Note: This option only applies to the IAR DLIB runtime environment.




To set related options, choose:

Project>Options>General Options>Library Configuration


--double

Syntax	<code>--double={32 64}</code>	
Parameters	32 (default)	32-bit doubles are used
	64	64-bit doubles are used
Description	Use this option to select the precision used by the compiler for representing the floating-point types <code>double</code> and <code>long double</code> . The compiler can use either 32-bit or 64-bit precision. By default, the compiler uses 32-bit precision.	
See also	Basic data types—floating-point types, page 299.	
		Project>Options>General Options>Target>Size of type 'double'

-e

Syntax	<code>-e</code>	
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 <code>-e</code> option and the <code>--strict</code> option cannot be used at the same time.	
See also	<i>Enabling language extensions</i> , page 195.	
		Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions
		Note: By default, this option is selected in the IDE.

--ec++

Syntax	<code>--ec++</code>	
Description	In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.	
		Project>Options>C/C++ Compiler>Language 1>C++ and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++

--eec++

Syntax `--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 202.



Project>Options>C/C++ Compiler>Language 1>C++

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Extended Embedded C++

--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 2>Enable multibyte support

--enable_restrict


Syntax `--enable_restrict`

Description Enables the Standard C keyword `restrict`. This option can be useful for improving analysis precision during optimization.




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

--error_limit

Syntax	<code>--error_limit=n</code>
Parameters	<p><i>n</i> The number of errors before the compiler stops the compilation. <i>n</i> must be a positive integer; 0 indicates no limit.</p>
Description	<p>Use the <code>--error_limit</code> option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.</p> <p> This option is not available in the IDE.</p>

-f

Syntax	<code>-f filename</code>
Parameters	See <i>Rules for specifying a filename or directory as parameters</i> , page 256.
Description	<p>Use this option to make the compiler read command line options from the named file, with the default filename extension <code>.xc1</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.</p> <p>Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.</p> <p> To set this option, use Project>Options>C/C++ Compiler>Extra Options.</p>

--guard_calls

Syntax	<code>--guard_calls</code>
Description	Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.
See also	<i>Managing a multithreaded environment</i> , page 152.

**Project>Options>C/C++ Compiler>Language 2>Guard Calls****--header_context**

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



This option is not available in the IDE.

--hw_workaround

Syntax	--hw_workaround=[CPU6 nop_after_lpm CPU40]	
Parameters	CPU6	Workaround for hardware issue CPU6
	nop_after_lpm	Workaround for hardware issues CPU18, CPU19, CPU24, CPU25, CPU27, and CPU29
	CPU40	Workaround for hardware issue CPU40
Description	Use this option to enable compiler workarounds for various hardware issues. Specify the option multiple times if you need to enable more than one workaround.	
See also	For more information about the workarounds available for the different hardware issues, see the release notes.	



When you select a device in the IAR Embedded Workbench IDE, the relevant hardware workarounds are enabled automatically.

Note: The workaround for hardware issue CPU6 is not enabled automatically. To set this option for CPU6, use **Project>Options>C/C++ Compiler>Extra Options**.

-I

Syntax	<code>-I path</code>
Parameters	<code>path</code> The search path for <code>#include</code> files
Description	Use this option to specify the search paths for <code>#include</code> files. This option can be used more than once on the command line.
See also	<i>Include file search procedure</i> , page 250.



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

-l

Syntax	<code>-l[a A b B c C D][N][H] {filename directory}</code>																		
Parameters	<table> <tr> <td>a (default)</td> <td>Assembler list file</td> </tr> <tr> <td>A</td> <td>Assembler list file with C or C++ source as comments</td> </tr> <tr> <td>b</td> <td>Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code>, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *</td> </tr> <tr> <td>B</td> <td>Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code>, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *</td> </tr> <tr> <td>c</td> <td>C or C++ list file</td> </tr> <tr> <td>C (default)</td> <td>C or C++ list file with assembler source as comments</td> </tr> <tr> <td>D</td> <td>C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values</td> </tr> <tr> <td>N</td> <td>No diagnostics in file</td> </tr> <tr> <td>H</td> <td>Include source lines from header files in output. Without this option, only source lines from the primary source file are included</td> </tr> </table>	a (default)	Assembler list file	A	Assembler list file with C or C++ source as comments	b	Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *	B	Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *	c	C or C++ list file	C (default)	C or C++ list file with assembler source as comments	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
a (default)	Assembler list file																		
A	Assembler list file with C or C++ source as comments																		
b	Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *																		
B	Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *																		
c	C or C++ list file																		
C (default)	C or C++ list file with assembler source as comments																		
D	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values																		
N	No diagnostics in file																		
H	Include source lines from header files in output. Without this option, only source lines from the primary source file are included																		

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

See also *Rules for specifying a filename or directory as parameters*, page 256.

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



To set related options, choose:

Project>Options>C/C++ Compiler>List

--library_module

Syntax --library_module

Description Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.



Project>Options>C/C++ Compiler>Output>Module type>Library Module

--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 do not define their 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 do not define their 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

--macro_positions_in_diagnostics

Syntax `--macro_positions_in_diagnostics`

Description Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.



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

--mfc

Syntax `--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 268, `--output`, `-o`, page 284, and *Multi-file compilation units*, page 235.



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++ 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 1>Enable IAR migration preprocessor extensions

--module_name

Syntax	<code>--module_name=<i>name</i></code>	
Parameters	<i>name</i>	An explicit object module name
Description	Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.	
	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	<code>--multiplier[=<i>[16 16s 32]</i>]</code>	
Parameters	No parameter	The 16-bit hardware multiplier
	16	The 16-bit hardware multiplier (same as no parameter)
	16s	The extended 16-bit 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 during hardware multiplier accesses.	

Note: You should also redirect library function calls to variants that use the hardware multiplier, see *Hardware multiplier support*, page 150.



To set related options, choose:

Project>Options>General Options>Target>Device

and

Project>Options>General Options>Target>Hardware multiplier

--no_call_frame_info

Syntax `--no_call_frame_info`

Description Normally, the compiler always generates call frame information in the output, to enable the debugger to display the call stack even in code from modules with no debug information. Use this option to disable the generation of call frame information.

See also *Call frame information*, page 189.



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

--no_code_motion

Syntax `--no_code_motion`

Description Use this option to disable code motion optimizations.

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

See also *Code motion*, page 238.



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

--no_cse

Syntax `--no_cse`

Description Use this option to disable common subexpression elimination.

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

See also

Common subexpression elimination, page 237.



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

--no_inline

Syntax

`--no_inline`

Description

Use this option to disable function inlining.

See also

Inlining functions, page 81.



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

--no_path_in_file_macros

Syntax

`--no_path_in_file_macros`

Description

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

See also

Description of predefined preprocessor symbols, page 362.



This option is not available in the IDE.

--no_rw_dynamic_init

Syntax

`--no_rw_dynamic_init`

Description

When the option `--ropi` is used, the actual addresses of constant variables are not known until at runtime. Therefore, pointers to constant data cannot be initialized statically. The compiler rewrites such static initializations to be dynamic where it is possible. Use the `--no_rw_dynamic_init` option to make the compiler give an error instead of rewriting the initializations.

This option has no effect if `--ropi` is not used.

See also

`--ropi`, page 288 and *Position-independent code and read-only data*, page 80.



Project>Options>C/C++ Compiler>Code>No dynamic read/write initialization

--no_size_constraints

Syntax `--no_size_constraints`

Description Use this option to relax the normal restrictions for code size expansion when optimizing for high speed.

Note: This option has no effect unless used with `-Ohs`.

See also *Speed versus size*, page 236.



Project>Options>C/C++ Compiler>Optimizations>Enable transformations>No size constraints

--no_static_destruction

Syntax `--no_static_destruction`

Description Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.

Use this option to suppress the emission of such code.

See also *System termination*, page 137.



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 268, `--dlib_config`, page 269, and `--system_include_dir`, page 291.



Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories

--no_tbaa

Syntax `--no_tbaa`

Description Use this option to disable type-based alias analysis.

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

See also *Type-based alias analysis*, page 238.



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

--no_typedefs_in_diagnostics

Syntax `--no_typedefs_in_diagnostics`

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

Example

```
typedef int (*MyPtr)(char const *);
MyPtr p = "My text string";
```

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_ubrof_messages

Syntax `--no_ubrof_messages`

Description Use this option to minimize the size of your application object file by excluding messages from the UBROF files. The file size can decrease by up to 60%. Note that the XLINK diagnostic messages will, however, be less useful when you use this option.



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

--no_unroll

Syntax `--no_unroll`

Description Use this option to disable loop unrolling.

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

See also *Loop unrolling*, page 238.



Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax `--no_warnings`

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



This option is not available in the IDE.

--no_wrap_diagnostics

Syntax `--no_wrap_diagnostics`

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



This option is not available in the IDE.

-O

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

Parameters

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

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

Description

Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level 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 234.



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

--omit_types

Syntax `--omit_types`

Description

By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.



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

--only_stdout

Syntax `--only_stdout`

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



This option is not available in the IDE.

--output, -o

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

Parameters See *Rules for specifying a filename or directory as parameters*, page 256.

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.

--pending_instantiations

Syntax `--pending_instantiations number`

Parameters *number* An integer that specifies the limit, where 64 is default. If 0 is used, there is no limit.

Description Use this option to specify the maximum number of instantiations of a given C++ template that is allowed to be in process of being instantiated at a given time. This is used for detecting recursive instantiations.



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

--predef_macros

Syntax	<code>--predef_macros {filename directory}</code>
Parameters	See <i>Rules for specifying a filename or directory as parameters</i> , page 256.
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 <code>predef</code> 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	<code>--preinclude includefile</code>
Parameters	See <i>Rules for specifying a filename or directory as parameters</i> , page 256.
Description	Use this option to make the compiler read the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.



Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

--preprocess

Syntax	<code>--preprocess [= [c] [n] [l]] {filename directory}</code>						
Parameters	<table> <tr> <td><code>c</code></td> <td>Preserve comments</td> </tr> <tr> <td><code>n</code></td> <td>Preprocess only</td> </tr> <tr> <td><code>l</code></td> <td>Generate <code>#line</code> directives</td> </tr> </table>	<code>c</code>	Preserve comments	<code>n</code>	Preprocess only	<code>l</code>	Generate <code>#line</code> directives
<code>c</code>	Preserve comments						
<code>n</code>	Preprocess only						
<code>l</code>	Generate <code>#line</code> directives						

See also *Rules for specifying a filename or directory as parameters*, page 256.

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



Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

Syntax	<code>--public_equ <i>symbol</i>[=<i>value</i>]</code>	
Parameters	<i>symbol</i>	The name of the assembler symbol to be defined
	<i>value</i>	An optional value of the defined assembler symbol
Description	This option is equivalent to defining a label in assembler language using the <code>EQU</code> directive and exporting it using the <code>PUBLIC</code> directive. This option can be used more than once on the command line.	



This option is not available in the IDE.

--reduce_stack_usage

Syntax	<code>--reduce_stack_usage</code>
Description	Use this option to make the compiler minimize the use of stack space at the cost of somewhat larger and slower code.



Project>Options>C/C++ Compiler>Code>Reduce stack usage

--regvar_r4

Syntax	<code>--regvar_R4</code>
Description	Use this option to reserve the register <code>R4</code> for use by global register variables, declared with the <code>__regvar</code> attribute. This can give more efficient code if used on frequently used variables.
See also	<code>--lock_r4</code> , page 275 and <code>__regvar</code> , page 319.



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 275 and `__regvar`, page 319.



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.



To set related options, choose:

Project>Options>C/C++ Compiler>Language 2>Floating-point semantics

--remarks

Syntax `--remarks`

Description The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also *Severity levels*, page 253.



Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax `--require_prototypes`

Description Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.



Project>Options>C/C++ Compiler>Language 1>Require prototypes

--ropi

Syntax `--ropi`

Description Use this option to make the compiler generate position-independent code and read-only data.

The `--ropi` option has some drawbacks and limitations, see *Drawbacks and limitations*, page 80.

Note: The `--ropi` option is deprecated in IAR Embedded Workbench for MSP430 from version 6.50.

See also

Position-independent code and read-only data, page 80, `__no_pic`, page 318, and `--no_rw_dynamic_init`, page 279. For information about the preprocessor symbol `__ROPI__`, see Description of predefined preprocessor symbols, page 362.



Project>Options>C/C++ Compiler>Code>Code and read-only data (ropi)

--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 321



Project>Options>C/C++ Compiler>Code>20-bit context save on interrupt

--segment

Syntax

```
--segment __memory_attribute=NEWSEGMENTBASENAME
--segment type=NEWSEGMENTNAME
```

Description

The compiler places functions and data objects into named segments which are referred to by the IAR XLINK Linker. Use the `--segment` option to perform one of these operations:

- To place data objects declared with the `__memory_attribute` in segments with names that begin with `NEWSEGMENTNAME`.
- To change the name of the special segments `SCSTACK`, `CODE`, `CODE16`, `RAMFUNC_CODE`, `DIFUNCT`, `ISR_CODE`, and `INTVEC` to a different name.

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 This command places the `__data16 int a;` defined variable in the `MYDATA_Z` segment:

```
--segment __data16=MYDATA
```

This command names the segment that contains interrupt vectors to `MYINTS` instead of the default name `INTVEC`:

```
--segment intvec=MYINTS
```

See also *Controlling data and function placement in memory*, page 230 and *Summary of extended keywords*, page 312.



Project>Options>C/C++ Compiler>Output>Override segment base name

--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 195.



Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict

--system_include_dir

Syntax `--system_include_dir path`

Parameters `path` The path to the system include files, see *Rules for specifying a filename or directory as parameters*, page 256.

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 268, `--dlib_config`, page 269, and `--no_system_include`, page 280.



This option is not available in the IDE.

--use_c++_inline

Syntax `--use_c++_inline`

Description Standard C uses slightly different semantics for the `inline` keyword than C++ does. Use this option if you want C++ semantics when you are using C.

See also *Inlining functions*, page 81



Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

--version

Syntax `--version`

Description Use this option to make the compiler send version information to the console and then exit.



This option is not available in the IDE.

--vla

Syntax	--vla
Description	Use this option to enable support for C99 variable length arrays. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the --c89 compiler option. Note: --vla should not be used together with the longjmp library function, as that can lead to memory leakages.
See also	<i>C language overview</i> , page 193.



Project>Options>C/C++ Compiler>Language 1>C dialect>Allow VLA

--warn_about_c_style_casts

Syntax	--warn_about_c_style_casts
Description	Use this option to make the compiler warn when C-style casts are used in C++ source code.



This option is not available in the IDE.

--warnings_affect_exit_code

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



This option is not available in the IDE.

--warnings_are_errors

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

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

See also

`--diag_warning`, page 267.



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

Data representation

- Alignment
- Basic data types—integer types
- Basic data types—floating-point types
- Pointer types
- Structure types
- Type qualifiers
- Data types in C++

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 303.

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 compiler ensures this by assigning an alignment to every data type, which means that the MSP430 microcontroller can read the data.

Basic data types—integer types

The compiler supports both all Standard C basic data types and some additional types.

INTEGER TYPES—AN OVERVIEW

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

Data type	Size	Range	Alignment
<code>bool</code>	8 bits	0 to 1	1
<code>char</code>	8 bits	0 to 255	1
<code>signed char</code>	8 bits	-128 to 127	1
<code>unsigned char</code>	8 bits	0 to 255	1
<code>signed short</code>	16 bits	-32768 to 32767	2
<code>unsigned short</code>	16 bits	0 to 65535	2
<code>signed int</code>	16 bits	-32768 to 32767	2
<code>unsigned int</code>	16 bits	0 to 65535	2
<code>signed long</code>	32 bits	-2^{31} to $2^{31}-1$	2
<code>unsigned long</code>	32 bits	0 to $2^{32}-1$	2
<code>signed long long</code>	64 bits	-2^{63} to $2^{63}-1$	2
<code>unsigned long long</code>	64 bits	0 to $2^{64}-1$	2

Table 33: Integer types

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

BOOL

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

THE LONG LONG TYPE

The `long long` data type is supported with one restriction:

The CLIB runtime library does not support the `long long` type.

THE ENUM TYPE

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

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

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

THE CHAR TYPE

The `char` type is by default `unsigned` in the compiler, but the `--char_is_signed` compiler option allows you to make it `signed`. Note, however, that the library is compiled with the `char` type as `unsigned`.

THE WCHAR_T TYPE

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

Note: The IAR CLIB Library has only rudimentary support for `wchar_t`.

BITFIELDS

In Standard C, `int`, `signed int`, and `unsigned int` can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (`char`, `short`, `int`, etc) results in a signed or unsigned bitfield.

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 signed.

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. This allocation scheme is referred to as the disjoint type bitfield allocation.

If you use the directive `#pragma bitfields=reversed`, bitfields are placed from the most significant bit to the least significant bit in each container. See bitfields, page 326.

Assume this example:

```
struct BitfieldExample
{
    uint32_t a : 12;
    uint16_t b : 3;
    uint16_t c : 7;
    uint8_t d;
};
```

The example in the disjoint types bitfield allocation strategy

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.

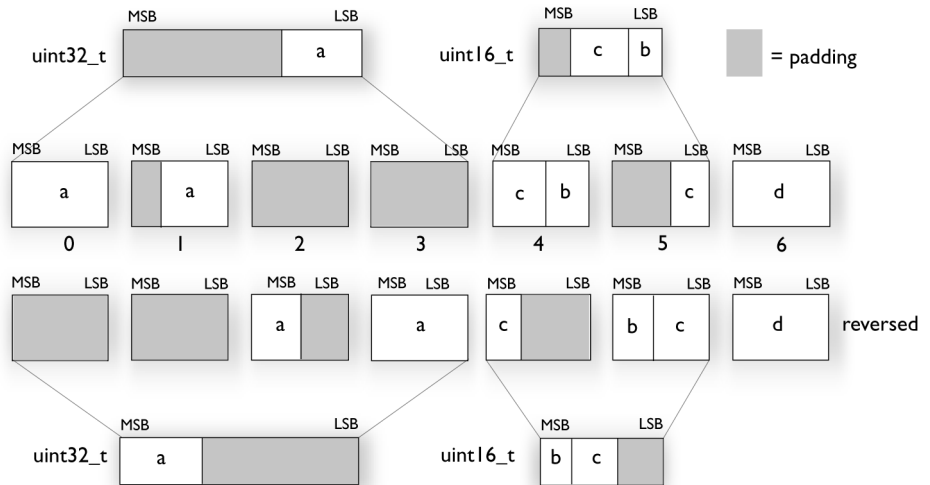
To place the second bitfield, `b`, a new container is allocated at offset 4, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

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`:



Basic data types—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:

Type	Size if double=32	Size if double=64
<code>float</code>	32 bits	32 bits
<code>double</code>	32 bits (default)	64 bits
<code>long double</code>	32 bits (default)	64 bits

Table 34: Floating-point types

The compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero.

FLOATING-POINT ENVIRONMENT

Exception flags are not supported. The `feraiseexcept` function does not raise any exceptions.

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 and for the MSP430X architecture when using the Small code model, function pointers are always 16 bits. For the MSP430X architecture when using the Large code model, function pointers are 20 bits and they can address the entire 1 Mbyte of memory. See this table:

Architecture	Data model	Code model	In register	In memory
MSP430	—	—	One 16-bit register	2 bytes
MSP430X	Any	Small	One 16-bit register	2 bytes
MSP430X	Small	Large	Two 16-bit registers	4 bytes
MSP430X	Medium/Large	Large	One 20-bit register	4 bytes

Table 35: Function pointers

Note: If ROPI is enabled, a normal function pointer holds the *linked* address of a function. The *actual* address is only computed when the function is called. You can use the function type attribute `__no_pic` to change this behavior, see `__no_pic`, page 318.

DATA POINTERS

These data pointers are available:

Keyword	Pointer size	Index type	Address range	In register	In memory
<code>__data16</code>	16 bits	signed int	0x0-0xFFFF	One 16-bit register	2 bytes
<code>__data20*</code>	20 bits	signed long	0x0-0xFFFFF	One 20-bit register	4 bytes

Table 36: Data pointers

*The `__data20` pointer type is not available for the MSP430X architecture when using the Small data model nor for the MSP430 architecture.

Note: A data pointer always holds the actual address of an object, regardless of whether ROPI is enabled or not.

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a value of an integer type to a pointer of a larger type is performed by zero extension
- Casting a pointer type to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension
- Casting from a larger pointer to a smaller pointer is performed by truncation.

size_t

`size_t` is the unsigned integer type of the result of the `sizeof` operator. For the MSP430 architecture, and for the MSP430X architecture in the Small and Medium data models, the type used for `size_t` is `unsigned int`. In the Large data model, the type used for `size_t` is `unsigned long int`.

ptrdiff_t

`ptrdiff_t` is the signed integer type of the result of subtracting two pointers. For the MSP430 architecture, and for the MSP430X architecture in the Small and Medium data models, the type used for `ptrdiff_t` is `signed int`. In the Large data model, the type used for `ptrdiff_t` is `signed long int`.

Note: It is sometimes possible to create an object that is so large that the result of subtracting two pointers in that object is negative. See this example:

```
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 type used for `intptr_t` is `signed int`. In the Large data model, the type used for `intptr_t` is `signed long int`.

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 OF STRUCTURE TYPES

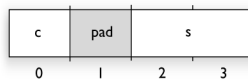
The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. Note that this alignment requirement also applies to a member that is a structure. To allow arrays of aligned structure objects, the size of a `struct` is adjusted to an even multiple of the alignment.

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).

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

This diagram shows the layout in memory:



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

PACKED STRUCTURE TYPES

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

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better

to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

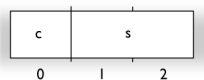
Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member,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

This example declares a packed structure:

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

#pragma pack()
```

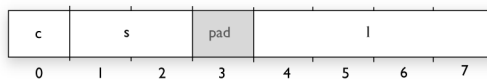
The structure `s` has this memory layout:



The next example declares a new non-packed structure, `S2`, that contains the structure `s` declared in the previous example:

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

The structure `S2` has this memory layout



The structure `s` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 2, which means that alignment of the structure `S2` will become 2.

For more information, see *Alignment of elements in a structure*, page 228.

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.

Note: For MSP430X, if the Large code model is used together with the Small data model, function pointers are accessed as 32-bit data.

DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, follow this example:

```
/* Header */
extern int const xVar;
#define x (*(int const volatile *) &xVar)

/* Source that uses x */
int DoSomething()
{
    return x;
}

/* Source that defines x */
#pragma segment = "FLASH"
int const xVar @ "FLASH" = 6;
```

The segment `FLASH` contains the initializers. They must be flashed manually when the application starts up.

Thereafter, the initializers can be reflashed with other values at any time.

DECLARING OBJECTS CONST

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

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

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

- General syntax rules for extended keywords
- Summary of extended keywords
- Descriptions of extended keywords

General syntax rules for extended keywords

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 more information about each attribute, see *Descriptions of extended keywords*, page 313. For information about how to use attributes to modify data, see the chapter *Data storage*.

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 270.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes explicitly in your declarations, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

Available *data memory attributes*:

```
__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 implicitly used by the compiler. You can specify one memory attribute for each level of pointer indirection.

General type attributes

Available *function type attributes* (affect how the function should be called):

```
__interrupt, __monitor, __task, __no_pic, __cc_version1, __cc_version2
```

Available *data type attributes*:

```
const, volatile
```

You can specify as many type attributes as required for each level of pointer indirection.

Syntax for type attributes used on data objects

Type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__data16 int i;
int __data16 j;
```

Both `i` and `j` are placed in `data16` memory.

Unlike `const` and `volatile`, when a type attribute is used before the type specifier in a derived type, the type attribute applies to the object, or typedef itself, except in structure member declarations.

```
int __data16 * p;      /* integer in data16 memory */
int * __data16 p;     /* pointer in data16 memory */
__data16 int * p;     /* pointer in data16 memory */
```

In all cases, if a memory attribute is not specified, an appropriate default memory type is used.

Using a type definition can sometimes make the code clearer:

```
typedef __data16 int d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in data16 memory. The variable `q1` can point to such integers.

You can also use the `#pragma type_attributes` directive to specify type attributes for a declaration. The type attributes specified in the pragma directive are applied to the data object or typedef being declared.

```
#pragma type_attribute=__data16
int * q2;
```

The variable `q2` is placed in data16 memory.

For more examples of using memory attributes, see *More examples*, page 64.

Syntax for type attributes used on functions

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

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

or

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

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

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

OBJECT ATTRIBUTES

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object. 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_alloc, __no_alloc16, __no_alloc_str, __no_alloc_str16,
__no_init, __persistent, __ro_placement
```

- Object attributes that can be used for functions and variables:

```
location, @, __root
```

- Object attributes that can be used for functions:

```
__intrinsic, __noreturn, __ramfunc, __raw, __save_reg20, 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 230. For more information about `vector`, see `vector`, page 345.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. This declaration is equivalent to the previous one:

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

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

Summary of extended keywords

This table summarizes the extended keywords:

Extended keyword	Description
<code>__cc_version1</code>	Specifies the Version1 calling convention
<code>__cc_version2</code>	Specifies the Version2 calling convention
<code>__data16</code>	Controls the storage of data objects
<code>__data20</code>	Controls the storage of data objects
<code>__interrupt</code>	Specifies interrupt functions
<code>__intrinsic</code>	Reserved for compiler internal use only
<code>__monitor</code>	Specifies atomic execution of a function
<code>__no_alloc,</code> <code>__no_alloc16</code>	Makes a constant available in the execution file

Table 37: Extended keywords summary

Extended keyword	Description
<code>__no_alloc_str,</code> <code>__no_alloc_str16</code>	Makes a string literal available in the execution file
<code>__no_init</code>	Places a data object in non-volatile memory
<code>__no_multiplier</code>	Suppresses generation of code that directly accesses the hardware multiplier
<code>__no_pic</code>	Controls how functions are called
<code>__noreturn</code>	Informs the compiler that the function will not return
<code>__persistent</code>	Ensures that a variable is only initialized, for example, by a code downloader, and not by <code>cstartup</code> .
<code>__ramfunc</code>	Makes a function execute in RAM
<code>__raw</code>	Prevents saving used registers in interrupt functions
<code>__regvar</code>	Permanently places a variable in a specified register
<code>__root</code>	Ensures that a function or variable is included in the object code even if unused
<code>__ro_placement</code>	Places <code>const volatile</code> data in read-only memory.
<code>__save_reg20</code>	Saves and restores all 20 bits in 20-bit registers
<code>__task</code>	Relaxes the rules for preserving registers

Table 37: Extended keywords summary (Continued)

Descriptions of extended keywords

This section gives detailed information about each extended keyword.

`__cc_version1`

Syntax	See <i>Syntax for type attributes used on functions</i> , page 311.
Description	The <code>__cc_version1</code> 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	<pre>__cc_version1 int func(int arg1, double arg2);</pre>
See also	<i>Calling convention</i> , page 179.

__cc_version2

Syntax	See <i>Syntax for type attributes used on functions</i> , page 311.
Description	The <code>__cc_version2</code> 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	<pre>__cc_version2 int func(int arg1, double arg2);</pre>
See also	<i>Calling convention</i> , page 179.

__data16

Syntax	See <i>Syntax for type attributes used on data objects</i> , page 310.
Description	The <code>__data16</code> 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 <code>__data16</code> attribute to create a pointer explicitly pointing to an object located in the data16 memory.
Storage information	<ul style="list-style-type: none"> ● Address range: 0-0xFFFF (64 Kbytes) ● Maximum object size: 65535 bytes ● Pointer size: 2 bytes
Example	<pre>__data16 int x;</pre>
See also	<i>Memory types (MSP430X only)</i> , page 60.

__data20

Syntax	See <i>Syntax for type attributes used on data objects</i> , page 310.
Description	<p>The <code>__data20</code> 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 <code>__data20</code> attribute to create a pointer explicitly pointing to an object located in the data20 memory.</p> <p>The <code>__data20</code> attribute cannot be used in the Small data model.</p>
Storage information	<ul style="list-style-type: none"> ● 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 60.

__interrupt

Syntax See *Syntax for type attributes used on functions*, page 311.

Description The `__interrupt` keyword specifies interrupt functions. To specify one or several interrupt vectors, use the `#pragma vector` directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a `void` return type and cannot have any parameters.

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



To make sure that the interrupt handler executes as fast as possible, you should compile it with `-Ohs`, or use `#pragma optimize=speed` if the module is compiled with another optimization goal.

Example

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

See also *Interrupt functions*, page 73, *vector*, page 345, and *INTVEC*, page 391.

__intrinsic

Description The `__intrinsic` keyword is reserved for compiler internal use only.

__monitor

Syntax See *Syntax for type attributes used on functions*, page 311.

Description The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.

Example `__monitor int get_lock(void);`

See also *Monitor functions*, page 76. For information about related intrinsic functions, see `__disable_interrupt`, page 353, `__enable_interrupt`, page 354, `__get_interrupt_state`, page 354, and `__set_interrupt_state`, page 358, respectively.

`__no_alloc, __no_alloc16`

Syntax See *Syntax for object attributes*, page 312.

Description Use the `__no_alloc` or `__no_alloc16` object attribute on a constant to make the constant available in the executable file without occupying any space in the linked application.

You cannot access the contents of such a constant from your application. You can take its address, which is an integer offset to the segment of the constant. The type of the offset is `unsigned long` when `__no_alloc` is used, and `unsigned short` when `__no_alloc16` is used.

Example `__no_alloc const struct MyData my_data @ "XXX" = {...};`

See also `__no_alloc_str`, `__no_alloc_str16`, page 316.

`__no_alloc_str, __no_alloc_str16`

Syntax `__no_alloc_str(string_literal @ segment)`

and

`__no_alloc_str16(string_literal @ segment)`

where

string_literal The string literal that you want to make available in the executable file.

segment The name of the segment to place the string literal in.

Description Use the `__no_alloc_str` or `__no_alloc_str16` operators to make string literals available in the executable file without occupying any space in the linked application.

The value of the expression is the offset of the string literal in the segment. For `__no_alloc_str`, the type of the offset is `unsigned long`. For `__no_alloc_str16`, the type of the offset is `unsigned short`.

Example

```
#define MYSEG "YYY"
#define X(str) __no_alloc_str(str @ MYSEG)

extern void dbg_printf(unsigned long fmt, ...)

#define DBGPRINTF(fmt, ...) dbg_printf(X(fmt), __VA_ARGS__)

void
foo(int i, double d)
{
    DBGPRINTF("The value of i is: %d, the value of d is: %f",i,d);
}
```

Depending on your debugger and the runtime support, this could produce trace output on the host computer. Note that there is no such runtime support in C-SPY, unless you use an external plugin module.

See also

`__no_alloc`, `__no_alloc16`, page 316.

__no_init**Syntax**

See *Syntax for object attributes*, page 312.

Description

Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Example

```
__no_init int myarray[10];
```

See also

Non-initialized variables, page 245.

__no_multiplier**Syntax**

See *Syntax for type attributes used on functions*, page 311.

Description

The `__no_multiplier` keyword suppresses generation of code that directly accesses the hardware multiplier. When using this keyword, the result is the same as when compiling without the `--multiplier` command line option.

Note: The linker might still redirect function calls, including calls to internal runtime library routines, to variants that use the hardware multiplier.

Example

```
__no_multiplier int mul(int x, int y)
{
    return x*y;
}
```

__no_pic

Syntax

See *Syntax for type attributes used on functions*, page 311.

Description

The `__no_pic` memory attribute allows position-independent code to call functions that are not position-independent. The attribute can only be used on function pointers and only when the `--ropi` option is used. Such function pointers are called with the normal calling sequence instead of with the position-independent one.

Example

```
int (__no_pic * fPtr)(int arg1, double arg2);
```

See also

Assembler instructions used for calling functions, page 186 and *Position-independent code and read-only data*, page 80.

__noreturn

Syntax

See *Syntax for object attributes*, page 312.

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`.

Note: At optimization levels medium or high, the `__noreturn` keyword might cause incorrect call stack debug information at any point where it can be determined that the current function cannot return.

Example

```
__noreturn void terminate(void);
```

__persistent

Syntax

See *Syntax for object attributes*, page 312.

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 `__persistent int x = 0;`

See also DATA16_P, page 386.

__ramfunc

Syntax See *Syntax for object attributes*, page 312.

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.

This attribute cannot be used if you have specified the `--ropi` option.

Example `__ramfunc int FlashPage(char * data, char * page);`

See also *Execution in RAM*, page 79.

__raw

Syntax See *Syntax for object attributes*, page 312.

Description This keyword prevents saving used registers in interrupt functions.

Example `__raw __interrupt void my_interrupt_function()`

__regvar

Syntax See *Syntax for type attributes used on functions*, page 311.

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 286 and `--regvar_r5`, page 287

__root

Syntax

See *Syntax for object attributes*, page 312.

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

For more information about modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide*.

__ro_placement

Syntax

See *Syntax for object attributes*, page 312.

Unlike most object attributes, when `--ropi` is enabled, the `__ro_placement` attribute must be specified both when a data object is defined and when it is declared.

Description

The `__ro_placement` attribute specifies that a data object should be placed in read-only memory. There are two cases where you might want to use this object attribute:

- Data objects declared `const volatile` are by default placed in read-write memory. Use the `__ro_placement` object attribute to place the data object in read-only memory instead.

- In C++, a data object declared `const` and that needs dynamic initialization is placed in read-write memory and initialized at system startup. If you use the `__ro_placement` object attribute, the compiler will give an error message if the data object needs dynamic initialization.

You can only use the `__ro_placement` object attribute on `const` objects.

In some cases (primarily involving simple constructors), the compiler will be able to optimize C++ dynamic initialization of a data object into static initialization. In that case no error message will be issued for the object.

Example

```
__ro_placement const volatile int x = 10;
```

__save_reg20

Syntax

See *Syntax for object attributes*, page 312.

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

```
__save_reg20 void myFunction(void);
```

See also

Interrupt functions for the MSP430X architecture, page 75.

__task

Syntax

See *Syntax for type attributes used on functions*, page 311.

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

```
__task void my_handler(void);
```

Pragma directives

- Summary of pragma directives
- Descriptions of pragma directives

Summary of 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 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.

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
<code>basic_template_matching</code>	Makes a template function fully memory-attribute aware.
<code>bis_nmi_ie1</code>	Allows non-maskable interrupt routines to enable interrupts.
<code>bitfields</code>	Controls the order of bitfield members.
<code>calls</code>	Lists possible called functions for indirect calls.
<code>call_graph_root</code>	Specifies that the function is a call graph root.
<code>constseg</code>	Places constant variables in a named segment.
<code>cstat_disable</code>	See the <i>C-STAT® Static Analysis Guide</i> .
<code>cstat_enable</code>	See the <i>C-STAT® Static Analysis Guide</i> .
<code>cstat_restore</code>	See the <i>C-STAT® Static Analysis Guide</i> .
<code>cstat_suppress</code>	See the <i>C-STAT® Static Analysis Guide</i> .
<code>data_alignment</code>	Gives a variable a higher (more strict) alignment.
<code>dataseg</code>	Places variables in a named segment.
<code>default_function_attributes</code>	Sets default type and object attributes for declarations and definitions of functions.

Table 38: Pragma directives summary

Pragma directive	Description
<code>default_variable_attributes</code>	Sets default type and object attributes for declarations and definitions of variables.
<code>diag_default</code>	Changes the severity level of diagnostic messages.
<code>diag_error</code>	Changes the severity level of diagnostic messages.
<code>diag_remark</code>	Changes the severity level of diagnostic messages.
<code>diag_suppress</code>	Suppresses diagnostic messages.
<code>diag_warning</code>	Changes the severity level of diagnostic messages.
<code>error</code>	Signals an error while parsing.
<code>include_alias</code>	Specifies an alias for an include file.
<code>inline</code>	Controls inlining of a function.
<code>language</code>	Controls the IAR Systems language extensions.
<code>location</code>	Specifies the absolute address of a variable, or places groups of functions or variables in named segments.
<code>message</code>	Prints a message.
<code>no_epilogue</code>	Uses a local return sequence
<code>object_attribute</code>	Adds object attributes to the declaration or definition of a variable or function.
<code>optimize</code>	Specifies the type and level of an optimization.
<code>pack</code>	Specifies the alignment of structures and union members.
<code>__printf_args</code>	Verifies that a function with a printf-style format string is called with the correct arguments.
<code>public_equ</code>	Defines a public assembler label and gives it a value.
<code>required</code>	Ensures that a symbol that is needed by another symbol is included in the linked output.
<code>rtmodel</code>	Adds a runtime model attribute to the module.
<code>__scanf_args</code>	Verifies that a function with a scanf-style format string is called with the correct arguments.
<code>section</code>	This directive is an alias for <code>#pragma segment</code> .
<code>segment</code>	Declares a segment name to be used by intrinsic functions.
<code>STDC CX_LIMITED_RANGE</code>	Specifies whether the compiler can use normal complex mathematical formulas or not.

Table 38: Pragma directives summary (Continued)

Pragma directive	Description
STDC FENV_ACCESS	Specifies whether your source code accesses the floating-point environment or not.
STDC FP_CONTRACT	Specifies whether the compiler is allowed to contract floating-point expressions or not.
vector	Specifies the vector of an interrupt or trap function.
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable.
type_attribute	Adds type attributes to a declaration or to definitions.

Table 38: Pragma directives summary (Continued)

Note: For portability reasons, see also Recognized pragma directives (6.10.6), page 413.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

basic_template_matching

Syntax	<code>#pragma basic_template_matching</code>
Description	Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications, see <i>Templates and data memory attributes</i> , page 210.
Example	<pre>// We assume that __data20 is the memory type of the default // pointer. #pragma basic_template_matching template<typename T> void fun(T *); void MyF() { fun((int __data16 *) 0); // T = int __data16 }</pre>

bis_nmi_ie1

Syntax	<code>#pragma bis_nmi_ie1=mask</code>
--------	---------------------------------------

See also *Bitfields*, page 298.

calls

Syntax `#pragma calls=function[, function...]`

Parameters *function* Any declared function

Description Use this pragma directive to list the functions that can be indirectly called in the following statement. This information can be used for stack usage analysis in the linker.

Note: For an accurate result, you must list all possible called functions.

Example

```
void Fun1(), Fun2();

void Caller(void (*fp)(void))
{
#pragma calls = Fun1, Fun2
  (*fp)();
}
```

See also *Stack usage analysis*, page 91

call_graph_root

Syntax `#pragma call_graph_root[=category]`

Parameters *category* A string that identifies an optional call graph root category

Description Use this pragma directive to specify that, for stack usage analysis purposes, the immediately following function is a call graph root. You can also specify an optional category. The compiler will usually automatically assign a call graph root category to interrupt and task functions. If you use the `#pragma call_graph_root` directive on such a function you will override the default category. You can specify any string as a category.

Example `#pragma call_graph_root="interrupt"`

See also *Stack usage analysis*, page 91

constseg

Syntax	<code>#pragma constseg=[<i>__memoryattribute</i>]{<i>SEGMENT_NAME</i> default}</code>						
Parameters	<table> <tr> <td><i>__memoryattribute</i></td> <td>An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.</td> </tr> <tr> <td><i>SEGMENT_NAME</i></td> <td>A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</td> </tr> <tr> <td>default</td> <td>Uses the default segment for constants.</td> </tr> </table>	<i>__memoryattribute</i>	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.	<i>SEGMENT_NAME</i>	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.	default	Uses the default segment for constants.
<i>__memoryattribute</i>	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.						
<i>SEGMENT_NAME</i>	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.						
default	Uses the default segment for constants.						
Description	This legacy pragma directive is supported for backward compatibility reasons. It can be used for placing 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 <code>#pragma constseg=default</code> directive.						
Example	<pre>#pragma constseg=__data20 MY_CONSTANTS const int factorySettings[] = {42, 15, -128, 0}; #pragma constseg=default</pre>						

data_alignment

Syntax	<code>#pragma data_alignment=<i>expression</i></code>		
Parameters	<table> <tr> <td><i>expression</i></td> <td>A constant which must be a power of two (1, 2, 4, etc.).</td> </tr> </table>	<i>expression</i>	A constant which must be a power of two (1, 2, 4, etc.).
<i>expression</i>	A constant which must be a power of two (1, 2, 4, etc.).		
Description	<p>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.</p> <p>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.</p> <p>Note: Normally, the size of a variable is a multiple of its alignment. The <code>data_alignment</code> 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.</p>		

dataseg

Syntax	<code>#pragma dataseg=[__memoryattribute]{SEGMENT_NAME default}</code>
Parameters	<p><code>__memoryattribute</code> An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.</p> <p><code>SEGMENT_NAME</code> A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</p> <p><code>default</code> Uses the default segment.</p>
Description	This legacy pragma directive is supported for backward compatibility reasons. It can be used for placing variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared <code>__no_init</code> . The setting remains active until you turn it off again with the <code>#pragma dataseg=default</code> directive.
Example	<pre>#pragma dataseg=__data16 MY_SECTIONSEGMENT __no_init char myBuffer[1000]; #pragma dataseg=default</pre>

default_function_attributes

Syntax	<pre>#pragma default_function_attributes=[attribute...]</pre> <p>where <i>attribute</i> can be:</p> <pre>type_attribute object_attribute @ segment_name</pre>
Parameters	<p><code>type_attribute</code> See <i>Type attributes</i>, page 309.</p> <p><code>object_attribute</code> See <i>Object attributes</i>, page 311.</p> <p><code>@ segment_name</code> See <i>Data and function placement in segments</i>, page 233.</p>
Description	Use this pragma directive to set default segment placement, type attributes, and object attributes for function declarations and definitions. The default settings are only used for

declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a `default_function_attributes` pragma directive with no attributes, restores the initial state where no such defaults have been applied to function declarations and definitions.

Example

```
/* Place following functions in segment MYSEG */
#pragma default_function_attributes = @ "MYSEG"
int fun1(int x) { return x + 1; }
int fun2(int x) { return x - 1; }
/* Stop placing functions into MYSEG */
#pragma default_function_attributes =
```

has the same effect as:

```
int fun1(int x) @ "MYSEG" { return x + 1; }
int fun2(int x) @ "MYSEG" { return x - 1; }
```

See also

location, page 335

object_attribute, page 337

type_attribute, page 344

default_variable_attributes

Syntax

```
#pragma default_variable_attributes=[ attribute...]
```

where *attribute* can be:

```
type_attribute
object_attribute
@ segment_name
```

Parameters

type_attribute See *Type attributes*, page 309.

object_attributes See *Object attributes*, page 311.

@ *segment_name* See *Data and function placement in segments*, page 233.

Description

Use this pragma directive to set default segment placement, type attributes, and object attributes for declarations and definitions of variables with static storage duration. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a `default_variable_attributes` pragma with no attributes restores the initial state of no such defaults being applied to variables with static storage duration.

Example

```
/* Place following variables in segment MYSEG */
#pragma default_variable_attributes = @ "MYSEG"
int var1 = 42;
int var2 = 17;
/* Stop placing variables into MYSEG */
#pragma default_variable_attributes =
```

has the same effect as:

```
int var1 @ "MYSEG" = 42;
int var2 @ "MYSEG" = 17;
```

See also

location, page 335

object_attribute, page 337

type_attribute, page 344

diag_default**Syntax**

```
#pragma diag_default=tag[, tag, ...]
```

Parameters

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

Description

Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also

Diagnostics, page 253.

diag_error**Syntax**

```
#pragma diag_error=tag[, tag, ...]
```

Parameters

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

Description Use this pragma directive to change the severity level to `error` for the specified diagnostics.

See also *Diagnostics*, page 253.

diag_remark

Syntax `#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 253.

diag_suppress

Syntax `#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 253.

diag_warning

Syntax `#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 253.

error

Syntax `#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

```
#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 `#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

```
#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 250.

inline

Syntax `#pragma inline[=forced|=never]`

Parameters

No parameter	Has the same effect as the <code>inline</code> keyword.
<code>forced</code>	Disables the compiler's heuristics and forces inlining.
<code>never</code>	Disables the compiler's heuristics and makes sure that the function will not be inlined.

Description

Use `#pragma inline` to advise the compiler that the function defined immediately after the directive should be inlined according to C++ inline semantics.

Specifying `#pragma inline=forced` will always inline the defined function. If the compiler fails to inline the function for some reason, for example due to recursion, a warning message is emitted.

Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` will enable inlining of the function also on the Medium optimization level.

See also *Inlining functions*, page 81.

language

Syntax `#pragma language={extended|default|save|restore}`

Parameters

<code>extended</code>	Enables the IAR Systems language extensions from the first use of the pragma directive and onward.
<code>default</code>	From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.

	<code>save restore</code>	Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code. Each use of <code>save</code> must be followed by a matching <code>restore</code> in the same file without any intervening <code>#include</code> directive.
Description	Use this pragma directive to control the use of language extensions.	
Example	<p>At the top of a file that needs to be compiled with IAR Systems extensions enabled:</p> <pre>#pragma language=extended /* The rest of the file. */</pre> <p>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:</p> <pre>#pragma language=save #pragma language=extended /* Part of source code. */ #pragma language=restore</pre>	
See also	<code>-e</code> , page 270 and <code>--strict</code> , page 290.	

location

Syntax	<code>#pragma location={<i>address</i> <i>NAME</i>}</code>	
Parameters	<i>address</i>	The absolute address of the global or static variable for which you want an absolute location.
	<i>NAME</i>	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
Description	Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either <code>__no_init</code> or <code>const</code> . Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive. Do not place variables that would normally be in different segments (for example, variables declared as <code>__no_init</code> and variables declared as <code>const</code>) in the same named segment.	

Example

```
#pragma location=0x22E
__no_init volatile char PORT1; /* PORT1 is located at address
                                0x22E */

#pragma segment="MY_SEG"
#pragma location="MY_SEG"
__no_init char PORT2; /* PORT2 is located in segment MY_SEG */

/* A better way is to use a corresponding mechanism */
#define MY_SEG _Pragma("location=\"MY_SEG\"")
/* ... */
MY_SEG __no_init int i; /* i is placed in the MY_SEG segment */
```

See also *Controlling data and function placement in memory*, page 230 and *Placing user-defined segments*, page 105.

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	<code>#pragma object_attribute=<i>object_attribute</i>[<i>object_attribute</i>...]</code>
Parameters	For information about object attributes that can be used with this pragma directive, see <i>Object attributes</i> , page 311.
Description	Use this pragma directive to add one or more IAR-specific object attributes to the declaration or definition of a variable or function. Object attributes affect the actual variable or function and not its type. When you define a variable or function, the union of the object attributes from all declarations including the definition, is used.
Example	<pre>#pragma object_attribute=__no_init char bar;</pre> <p>is equivalent to:</p> <pre>__no_init char bar;</pre>
See also	<i>General syntax rules for extended keywords</i> , page 309.

optimize

Syntax	<code>#pragma optimize=[<i>goal</i>] [<i>level</i>] [<i>no_optimization</i>...]</code>				
Parameters	<table> <tr> <td><i>goal</i></td> <td> <p>Choose between:</p> <ul style="list-style-type: none"> <i>size</i>, optimizes for size <i>balanced</i>, optimizes balanced between speed and size <i>speed</i>, optimizes for speed. <i>no_size_constraints</i>, optimizes for speed, but relaxes the normal restrictions for code size expansion. </td> </tr> <tr> <td><i>level</i></td> <td>Specifies the level of optimization; choose between none, low, medium, or high.</td> </tr> </table>	<i>goal</i>	<p>Choose between:</p> <ul style="list-style-type: none"> <i>size</i>, optimizes for size <i>balanced</i>, optimizes balanced between speed and size <i>speed</i>, optimizes for speed. <i>no_size_constraints</i>, optimizes for speed, but relaxes the normal restrictions for code size expansion. 	<i>level</i>	Specifies the level of optimization; choose between none, low, medium, or high.
<i>goal</i>	<p>Choose between:</p> <ul style="list-style-type: none"> <i>size</i>, optimizes for size <i>balanced</i>, optimizes balanced between speed and size <i>speed</i>, optimizes for speed. <i>no_size_constraints</i>, optimizes for speed, but relaxes the normal restrictions for code size expansion. 				
<i>level</i>	Specifies the level of optimization; choose between none, low, medium, or high.				

`no_optimization` Disables one or several optimizations; choose between:

- `no_code_motion`, disables code motion
- `no_crosscall`, disables interprocedural cross call
- `no_crossjump`, disables interprocedural cross jump
- `no_cse`, disables common subexpression elimination
- `no_inline`, disables function inlining
- `no_tbaa`, disables type-based alias analysis
- `no_unroll`, disables loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `size`, `balanced`, `speed`, and `no_size_constraints` only have effect on the `high` optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=speed
int SmallAndUsedOften()
{
    /* Do something here. */
}

#pragma optimize=size
int BigAndSeldomUsed()
{
    /* Do something here. */
}
```

See also

Fine-tuning enabled transformations, page 237.

pack

Syntax	<pre>#pragma pack(<i>n</i>) #pragma pack() #pragma pack({push pop}[,<i>name</i>][,<i>n</i>])</pre>										
Parameters	<table> <tr> <td><i>n</i></td> <td>Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16</td> </tr> <tr> <td>Empty list</td> <td>Restores the structure alignment to default</td> </tr> <tr> <td><i>push</i></td> <td>Sets a temporary structure alignment</td> </tr> <tr> <td><i>pop</i></td> <td>Restores the structure alignment from a temporarily pushed alignment</td> </tr> <tr> <td><i>name</i></td> <td>An optional pushed or popped alignment label</td> </tr> </table>	<i>n</i>	Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16	Empty list	Restores the structure alignment to default	<i>push</i>	Sets a temporary structure alignment	<i>pop</i>	Restores the structure alignment from a temporarily pushed alignment	<i>name</i>	An optional pushed or popped alignment label
<i>n</i>	Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16										
Empty list	Restores the structure alignment to default										
<i>push</i>	Sets a temporary structure alignment										
<i>pop</i>	Restores the structure alignment from a temporarily pushed alignment										
<i>name</i>	An optional pushed or popped alignment label										
Description	<p>Use this pragma directive to specify the maximum alignment of <code>struct</code> and <code>union</code> members.</p> <p>The <code>#pragma pack</code> directive affects declarations of structures following the pragma directive to the next <code>#pragma pack</code> or the end of the compilation unit.</p> <p>Note: This can result in significantly larger and slower code when accessing members of the structure.</p>										
See also	<i>Structure types</i> , page 303.										

public_equ

Syntax	<pre>#pragma public_equ="symbol", value</pre>				
Parameters	<table> <tr> <td><i>symbol</i></td> <td>The name of the assembler symbol to be defined (string).</td> </tr> <tr> <td><i>value</i></td> <td>The value of the defined assembler symbol (integer constant expression).</td> </tr> </table>	<i>symbol</i>	The name of the assembler symbol to be defined (string).	<i>value</i>	The value of the defined assembler symbol (integer constant expression).
<i>symbol</i>	The name of the assembler symbol to be defined (string).				
<i>value</i>	The value of the defined assembler symbol (integer constant expression).				
Description	Use this pragma directive to define a public assembler label and give it a value.				
Example	<pre>#pragma public_equ="MY_SYMBOL", 0x123456</pre>				
See also	<i>--public_equ</i> , page 286.				

__printf_args

Syntax	<code>#pragma __printf_args</code>
Description	Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct.
Example	<pre>#pragma __printf_args int printf(char const *,...); void PrintNumbers(unsigned short x) { printf("%d", x); /* Compiler checks that x is an integer */ }</pre>

required

Syntax	<code>#pragma required=<i>symbol</i></code>
Parameters	<i>symbol</i> Any statically linked function or variable.
Description	<p>Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.</p> <p>Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.</p>
Example	<pre>const char copyright[] = "Copyright by me"; #pragma required=copyright int main() { /* Do something here. */ }</pre> <p>Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.</p>
See also	<i>Inline assembler</i> , page 169

rtmodel

Syntax	<code>#pragma rtmodel="key", "value"</code>
Parameters	<p><code>"key"</code> A text string that specifies the runtime model attribute.</p> <p><code>"value"</code> A text string that specifies the value of the runtime model attribute. Using the special value <code>*</code> is equivalent to not defining the attribute at all.</p>
Description	<p>Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.</p> <p>This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value <code>*</code>. It can, however, be useful to state explicitly that the module can handle any runtime model.</p> <p>A module can have several runtime model definitions.</p> <p>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.</p>
Example	<pre>#pragma rtmodel="I2C", "ENABLED"</pre> <p>The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.</p>
See also	<i>Checking module consistency</i> , page 111.

__scanf_args

Syntax	<code>#pragma __scanf_args</code>
Description	Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example <code>%d</code>) is syntactically correct.

Example

```
#pragma __scanf_args
int scanf(char const *,...);

int GetNumber()
{
    int nr;
    scanf("%d", &nr); /* Compiler checks that
                       the argument is a
                       pointer to an integer */

    return nr;
}
```

segment

Syntax

```
#pragma segment="NAME" [__memoryattribute]
alias
#pragma section="NAME" [__memoryattribute] [align]
```

Parameters

<i>NAME</i>	The name of the segment.
<i>__memoryattribute</i>	An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.
<i>align</i>	Specifies an alignment for the segment. The value must be a constant integer expression to the power of two.

Description

Use this pragma directive to define a segment name that can be used by the segment operators `__segment_begin`, `__segment_end`, and `__segment_size`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

The *align* and the *__memoryattribute* parameters are only relevant when used together with the segment operators `__segment_begin`, `__segment_end`, and `__segment_size`. If you consider using *align* on an individual variable to achieve a higher alignment, you must instead use the `#pragma data_alignment` directive.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

```
void __memoryattribute *.
```

Note: To place variables or functions in a specific segment, use the `#pragma location` directive or the `@` operator.

Example `#pragma segment="MYDATA16" __data16 4`

See also *Dedicated segment operators*, page 197 and the chapters *Linking overview* and *Linking your application*.

STDC CX_LIMITED_RANGE

Syntax `#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}`

Parameters

ON	Normal complex mathematic formulas can be used.
OFF	Normal complex mathematic formulas cannot be used.
DEFAULT	Sets the default behavior, that is OFF.

Description

Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for `*` (multiplication), `/` (division), and `abs`.

Note: This directive is required by Standard C. The directive is recognized but has no effect in the compiler.

STDC FENV_ACCESS

Syntax `#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}`

Parameters

ON	Source code accesses the floating-point environment. Note that this argument is not supported by the compiler.
OFF	Source code does not access the floating-point environment.
DEFAULT	Sets the default behavior, that is OFF.

Description

Use this pragma directive to specify whether your source code accesses the floating-point environment or not.

Note: This directive is required by Standard C.

STDC FP_CONTRACT

Syntax	<code>#pragma STDC FP_CONTRACT {ON OFF DEFAULT}</code>	
Parameters	ON	The compiler is allowed to contract floating-point expressions.
	OFF	The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler.
	DEFAULT	Sets the default behavior, that is ON.
Description	Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C.	
Example	<code>#pragma STDC FP_CONTRACT=ON</code>	

type_attribute

Syntax	<code>#pragma type_attribute=type_attr[type_attr...]</code>	
Parameters	For information about type attributes that can be used with this pragma directive, see <i>Type attributes</i> , page 309.	
Description	Use this pragma directive to specify IAR-specific <i>type attributes</i> , which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects. This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.	
Example	In this example, an <code>int</code> object with the memory attribute <code>__data16</code> is defined: <code>#pragma type_attribute=__data16 int x;</code> This declaration, which uses extended keywords, is equivalent: <code>__data16 int x;</code>	
See also	The chapter <i>Extended keywords</i> .	

unroll

Syntax	<code>#pragma unroll=<i>n</i></code>
Parameters	<i>n</i> The number of loop bodies in the unrolled loop, a constant integer. <code>#pragma unroll = 1</code> will prevent the unrolling of a loop.
Description	<p>Use this pragma directive to specify that the loop following immediately after the directive should be unrolled and that the unrolled loop should have <i>n</i> copies of the loop body. The pragma directive can only be placed immediately before a <code>for</code>, <code>do</code>, or <code>while</code> loop, whose number of iterations can be determined at compile time.</p> <p>Normally, unrolling is most effective for relatively small loops. However, in some cases, unrolling larger loops can be beneficial if it exposes opportunities for further optimizations between the unrolled loop iterations, for example common subexpression elimination or dead code elimination.</p> <p>The <code>#pragma unroll</code> directive can be used to force a loop to be unrolled if the unrolling heuristics are not aggressive enough. The pragma directive can also be used to reduce the aggressiveness of the unrolling heuristics.</p>
Example	<pre>#pragma unroll=4 for (i = 0; i < 64; ++i) { foo(i * k, (i + 1) * k); }</pre>
See also	<i>Loop unrolling</i> , page 238

vector

Syntax	<code>#pragma vector=<i>vector1</i>[, <i>vector2</i>, <i>vector3</i>, ...]</code>
Parameters	<i>vectorN</i> The vector number(s) of an interrupt function.
Description	<p>Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.</p> <p>This pragma directive cannot be used if you have specified the <code>--ropi</code> option.</p>

Example `#pragma vector=
__interrupt void my_handler(void);`

weak

Syntax `#pragma weak symbol1[=symbol2]`

Parameters

symbol1 A function or variable with external linkage.
symbol2 A defined function or variable.

Description

This pragma directive can be used in one of two ways:

- To make the definition of a function or variable with external linkage a weak definition.
- To create a weak alias for another function or variable. You can make more than one alias for the same function or variable.

Example

To make the definition of `foo` a weak definition, write:

```
#pragma weak foo
```

To make `NMI_Handler` a weak alias for `Default_Handler`, write:

```
#pragma weak NMI_Handler=Default_Handler
```

If `NMI_Handler` is not defined elsewhere in the program, all references to `NMI_Handler` will refer to `Default_Handler`.

Intrinsic functions

- Summary of intrinsic functions
- Descriptions of intrinsic functions

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

```
__disable_interrupt
```

This table summarizes the intrinsic functions:

Intrinsic function	Description
<code>__bcd_add_type</code>	Performs a binary coded decimal operation
<code>__bic_SR_register</code>	Clears bits in the SR register
<code>__bic_SR_register_on_exit</code>	Clears bits in the SR register when an interrupt or monitor function returns
<code>__bis_GIE_interrupt_state</code>	Sets the GIE bit of the processor status register
<code>__bis_SR_register</code>	Sets bits in the SR register
<code>__bis_SR_register_on_exit</code>	Sets bits in the SR register when an interrupt or monitor function returns
<code>__code_distance</code>	Returns the distance between the code and the position it was linked for
<code>__data16_read_addr</code>	Reads data to a 20-bit SFR register
<code>__data16_write_addr</code>	Writes data to a 20-bit SFR register
<code>__data20_longjmp</code>	Like the standard function <code>longjmp</code> , but allows the jump buffer to be placed in data20 memory
<code>__data20_read_type</code>	Reads data which has a 20-bit address
<code>__data20_setjmp</code>	Like the standard function <code>setjmp</code> , but allows the jump buffer to be placed in data20 memory
<code>__data20_write_type</code>	Writes data which has a 20-bit address

Table 39: Intrinsic functions summary

Intrinsic function	Description
<code>__delay_cycles</code>	Provides cycle-accurate delay functionality
<code>__disable_interrupt</code>	Disables interrupts
<code>__enable_interrupt</code>	Enables interrupts
<code>__even_in_range</code>	Makes switch statements rely on the specified value being even and within the specified range
<code>__get_interrupt_state</code>	Returns the interrupt state
<code>__get_R4_register</code>	Returns the value of the R4 register
<code>__get_R5_register</code>	Returns the value of the R5 register
<code>__get_SP_register</code>	Returns the value of the stack pointer
<code>__get_SR_register</code>	Returns the value of the SR register
<code>__get_SR_register_on_exit</code>	Returns the value that the processor status register will have when the current interrupt or monitor function returns
<code>__low_power_mode_n</code>	Enters a MSP430 low power mode
<code>__low_power_mode_off_on_exit</code>	Turns off low power mode when a monitor or interrupt function returns
<code>__no_operation</code>	Inserts a NOP instruction
<code>__op_code</code>	Inserts a constant into the instruction stream
<code>__saturated_add_signed_type</code>	Performs a saturated addition
<code>__saturated_add_unsigned_type</code>	Performs a saturated addition
<code>__saturated_sub_signed_type</code>	Performs a saturated subtraction
<code>__saturated_sub_unsigned_type</code>	Performs a saturated subtraction
<code>__set_interrupt_state</code>	Restores the interrupt state
<code>__set_R4_register</code>	Writes a specific value to the R4 register
<code>__set_R5_register</code>	Writes a specific value to the R5 register
<code>__set_SP_register</code>	Writes a specific value to the SP register
<code>__swap_bytes</code>	Executes the SWPB instruction

Table 39: Intrinsic functions summary (Continued)

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

`__bcd_add_type`

Syntax

```
unsigned type __bcd_add_type(unsigned type x, unsigned type y);
```

where:

`type` Can be one of the types `short`, `long`, or `long long`

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:

Function	Return value
<code>__bcd_add_short</code>	Returns the sum of the two <code>short</code> parameters. The parameters and the return value are represented as four-digit BCD numbers.
<code>__bcd_add_long</code>	Returns the sum of the two <code>long</code> parameters. The parameters and the return value are represented as eight-digit BCD numbers.
<code>__bcd_add_long_long</code>	Returns the sum of the two <code>long long</code> parameters. The parameters and the return value are represented as sixteen-digit BCD numbers.

Table 40: 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	<code>void __bic_SR_register_on_exit(unsigned short);</code>
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_GIE_interrupt_state

Syntax	<code>void __bis_GIE_interrupt_state(__istate_t)</code>
Description	Sets the GIE bit of the processor status register, if set in the state.
Example	<pre>__istate_t state = __get_interrupt_state(); __disable_interrupt(); ... __bis_GIE_interrupt_state(state);</pre>

__bis_SR_register

Syntax	<code>void __bis_SR_register(unsigned short);</code>
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	<code>void __bis_SR_register_on_exit(unsigned short);</code>
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.

__code_distance

Syntax	<code>long __code_distance(void);</code>
--------	--

Description Returns the number of bytes between the placement of code in memory and the position it was linked for. For non-position-independent code, the function returns 0.

The actual start address of a position-independent segment `MY_SEG` is `(char *) __segment_begin("MY_SEG") + __code_distance()`.

See also *Position-independent code and read-only data*, page 80 and *Dedicated segment operators*, page 197

__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.



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

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.



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_longjmp

Syntax `void __data20_longjmp(__JMP_BUF_ELEMENT_TYPE__ __data20 *, int);`

Description Like the standard function `longjmp`, but allows you to place the jump buffer in data20 memory.

__data20_read_type

Syntax `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:

Function	Operation size	Alignment
<code>unsigned char __data20_read_char</code>	1 byte	1
<code>unsigned short __data20_read_short</code>	2 bytes	2
<code>unsigned long __data20_read_long</code>	4 bytes	2

Table 41: 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_setjmp

Syntax `int __data20_setjmp(__JMP_BUF_ELEMENT_TYPE__ __data20 *);`

Description Like the standard function `setjmp`, but allows the jump buffer to be placed in data20 memory.

__data20_write_type

Syntax `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`

Description Writes 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:

Function	Operation size	Alignment
unsigned char <code>__data20_write_char</code>	1 byte	1
unsigned short <code>__data20_write_short</code>	2 bytes	2
unsigned long <code>__data20_write_long</code>	4 bytes	2

Table 42: 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 instruction.

__enable_interrupt

Syntax `void __enable_interrupt(void);`

Description Enables interrupts by inserting the `__enable_interrupt` 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 74.

__get_interrupt_state

Syntax `__istate_t __get_interrupt_state(void);`

Description

Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

Example

```
#include "intrinsics.h"

void CriticalFn()
{
    __istate_t s = __get_interrupt_state();
    __disable_interrupt();

    /* Do something here. */

    __set_interrupt_state(s);
}
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

__get_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 275.

__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 275.

__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 `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 `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 `void __no_operation(void);`

Description Inserts a NOP instruction.

__op_code

Syntax	<code>void __op_code(unsigned short);</code>
Description	Emits the 16-bit value into the instruction stream for the current function by inserting a DC16 constant.

__saturated_add_signed_type

Syntax	<code>__saturated_add_signed_type(signed type, signed type);</code> where <i>type</i> can be one of <code>char</code> , <code>short</code> , <code>long</code> , and <code>long long</code>
Description	Performs a saturated addition. When the mathematical result is outside the range of the destination type, the result is the smallest or largest value that the type can hold.
Example	<code>__saturated_add_signed_char(100, 100)</code> returns 127

__saturated_add_unsigned_type

Syntax	<code>__saturated_add_unsigned_type(unsigned type, unsigned type);</code> where <i>type</i> can be one of <code>char</code> , <code>short</code> , <code>long</code> , and <code>long long</code>
Description	Performs a saturated addition. When the mathematical result is outside the range of the destination type, the result is the smallest or largest value that the type can hold.
Example	<code>__saturated_add_unsigned_char(200, 200)</code> returns 255

__saturated_sub_signed_type

Syntax	<code>__saturated_sub_signed_type(signed type, signed type);</code> where <i>type</i> can be one of <code>char</code> , <code>short</code> , <code>long</code> , and <code>long long</code>
Description	Performs a saturated subtraction. When the mathematical result is outside the range of the destination type, the result is the smallest or largest value that the type can hold.
Example	<code>__saturated_sub_signed_char(100, 120)</code> returns -20 <code>__saturated_sub_signed_char(-100, 120)</code> returns -128

__saturated_sub_unsigned_type

Syntax	<code>__saturated_sub_unsigned_type(unsigned type, unsigned type);</code> where <i>type</i> can be one of <code>char</code> , <code>short</code> , <code>long</code> , and <code>long long</code>
Description	Performs a saturated subtraction. When the mathematical result is outside the range of the destination type, the result is the smallest or largest value that the type can hold.
Example	<code>__saturated_sub_unsigned_char(200, 100)</code> returns 100 <code>__saturated_sub_unsigned_char(100, 200)</code> returns 0

__set_interrupt_state

Syntax	<code>void __set_interrupt_state(__istate_t);</code>
Description	Restores the interrupt state to a value previously returned by the <code>__get_interrupt_state</code> function. The intrinsic function <code>__set_interrupt_state()</code> restores the entire status register. This includes the Global Interrupt Enable (GIE) bit as well as the bits <code>SCGX</code> , <code>OSC_OFF</code> , and <code>CPU_OFF</code> . For information about the <code>__istate_t</code> type, see <code>__get_interrupt_state</code> , page 354.
See also	<code>__bis_GIE_interrupt_state</code> , page 350

__set_R4_register

Syntax	<code>void __set_R4_register(unsigned short);</code>
Description	Writes a specific value to the R4 register. This intrinsic function is only available when R4 is locked.
See also	<code>--lock_r4</code> , page 275.

__set_R5_register

Syntax	<code>void __set_R5_register(unsigned short);</code>
Description	Writes a specific value to the R5 register. This intrinsic function is only available when R5 is locked.

See also `--lock_r5`, page 275.

__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

- Overview of the preprocessor
- Description of predefined preprocessor symbols
- Descriptions of miscellaneous preprocessor extensions

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 more information, see *Description of predefined preprocessor symbols*, page 362.
- 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 263.
- Preprocessor extensions
There are several preprocessor extensions, for example many `pragma` directives; for more information, see the chapter *Pragma directives*. For information about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 366.
- Preprocessor output
Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 285.

To specify a path for an include file, use forward slashes:

```
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile", "rt");
```

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.

Description of predefined preprocessor symbols

This section lists and describes the preprocessor symbols.

__BASE_FILE__

Description A string that identifies the name of the base source file (that is, not the header file), being compiled.

See also `__FILE__`, page 363, and `--no_path_in_file_macros`, page 279.

__BUILD_NUMBER__

Description A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.

__CODE_MODEL__

Description An integer that identifies the code model in use. The value reflects the setting of the `--code_model` option and is defined to `__CODE_MODEL_SMALL__` or `__CODE_MODEL_LARGE__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.

__CORE__

Description An integer that identifies the chip core in use. The value reflects the setting of the `--core` option and is defined to `__430__` for the MSP430 architecture and to `__430X__` for the MSP430X architecture. These symbolic names can be used when testing the `__CORE__` symbol.

__COUNTER__

Description A macro that expands to a new integer each time it is expanded, starting at zero (0) and counting up.

__cplusplus

Description An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with

`#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

__DATA_MODEL__

Description

An integer that identifies the data model in use. The value reflects the setting of the `--data_model` option and is defined to `__DATA_MODEL_SMALL__`, `__DATA_MODEL_MEDIUM__`, or `__DATA_MODEL_LARGE__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.

__DATE__

Description

A string that identifies the date of compilation, which is returned in the form "Mmm dd yy", for example "Oct 30 2014"

This symbol is required by Standard C.

__embedded_cplusplus

Description

An integer which is defined to 1 when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

__FILE__

Description

A string that identifies the name of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

See also

`__BASE_FILE__`, page 362, and `--no_path_in_file_macros`, page 279.

__func__

Description A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

This symbol is required by Standard C.

See also *-e*, page 270 and `__PRETTY_FUNCTION__`, page 365.

__FUNCTION__

Description A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

See also *-e*, page 270 and `__PRETTY_FUNCTION__`, page 365.

__IAR_SYSTEMS_ICC__

Description An integer that identifies the IAR compiler platform. The current value is 8. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.

__ICC430__

Description An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for MSP430.

__LINE__

Description An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

__POSITION_INDEPENDENT_CODE__

Description An integer that is set to 1 when the code is compiled with the option `--ropi`.

__PRETTY_FUNCTION__

Description A predefined string identifier that is initialized with the function name, including parameter types and return type, of the function in which the symbol is used, for example "void func(char)". This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

See also -e, page 270 and `__func__`, page 364.

__REGISTER_MODEL__

Description An integer that equals one of the following: `__REGISTER_MODEL_REG16__` (for MSP430 and MSP430X in the Small data model) or `__REGISTER_MODEL_REG20__` (for MSP430X in the Medium and Large data models).

__REGISTER_R4__

Description An integer that identifies the use of register R4. The value reflects the setting of the `--lock_r4` and `--regvar_r4` options and is defined to `__REGISTER_FREE__`, `__REGISTER_LOCKED__`, or `__REGISTER_REGVAR__`.

__REGISTER_R5__

Description An integer that identifies the use of register R5. The value reflects the setting of the `--lock_r5` and `--regvar_r5` options and is defined to `__REGISTER_FREE__`, `__REGISTER_LOCKED__`, or `__REGISTER_REGVAR__`.

__ROPI__

Description An integer that is set to 1 when the code is compiled with the option `--ropi`.

__STDC__

Description An integer that is set to 1, which means the compiler adheres to Standard C. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to Standard C.* This symbol is required by Standard C.

__STDC_VERSION__

Description An integer that identifies the version of the C standard in use. The symbol expands to 199901L, unless the `--c89` compiler option is used in which case the symbol expands to 199409L. This symbol does not apply in EC++ mode.

This symbol is required by Standard C.

__SUBVERSION__

Description An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.

__TIME__

Description A string that identifies the time of compilation in the form "hh:mm:ss".

This symbol is required by Standard C.

__TIMESTAMP__

Description A string constant that identifies the date and time of the last modification of the current source file. The format of the string is the same as that used by the `asctime` standard function (in other words, "Tue Sep 16 13:03:52 2014").

__VER__

Description An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: (100 * the major version number + the minor version number). For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and Standard C directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

See also

`_ReportAssert`, page 144.

#warning message

Syntax

```
#warning message
```

where *message* can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.

C/C++ standard library functions

- C/C++ standard library overview
- DLIB runtime environment—implementation details
- CLIB runtime environment—implementation details

For detailed reference information about the library functions, see the online help system.

C/C++ standard library overview

The compiler comes with two different implementations of the C/C++ standard library:

The IAR DLIB Runtime Environment is a complete implementation of the C/C++ standard library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.

The IAR CLIB Runtime Environment is a light-weight implementation of the C standard library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format and it does not support C++. Note that the legacy CLIB runtime environment 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 more information about customization, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For more information about library functions, see the chapter *Implementation-defined behavior for Standard C* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic project configuration*, page 54. The linker will include only those routines that are required—directly or indirectly—by your application.

See also *Overriding library modules*, page 123 for information about how you can override library modules with your own versions.

ALTERNATIVE MORE ACCURATE LIBRARY FUNCTIONS

The default implementation of `cos`, `sin`, `tan`, and `pow` is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named `__iar_xxx_accuratef` for float variants of the functions and `__iar_xxx_accuratel` for long double variants of the functions, and where `xxx` is `cos`, `sin`, etc.

To use any of these more accurate versions, use the `-e` linker option.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB runtime environment are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcstomb`, `wctomb`

- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `scanf`, `sscanf`, `getchar`, and `putchar`. In addition, if you are using the options `--enable_multibyte` and `--dlib_config=Full`, the `printf` and `sprintf` functions (or any variants) can also use the heap.

For the CLIB library, the `qsort` function and functions that use files in some way are non-reentrant. This includes `printf`, `scanf`, `getchar`, and `putchar`. However, the functions `sprintf` and `sscanf` are reentrant.

Functions that can set `errno` are not reentrant, because an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. This applies to math and string conversion functions, among others.

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

THE LONGJMP FUNCTION



A `longjmp` is in effect a jump to a previously defined `setjmp`. Any variable length arrays or C++ objects residing on the stack during stack unwinding will not be destroyed. This can lead to resource leaks or incorrect application behavior.

DLIB runtime environment—implementation details

The DLIB runtime environment provides most of the important C and C++ standard library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior for Standard C* in this guide.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code, see the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of MSP430 features. See the chapter *Intrinsic functions* for more information.

In addition, the DLIB runtime environment includes some added C functionality, see *Added C functionality*, page 375.

C HEADER FILES

This section lists the C header files specific to the DLIB runtime environment. Header files may additionally contain target-specific definitions; these are documented in the chapter *Using C*.

This table lists the C header files:

Header file	Usage
<code>assert.h</code>	Enforcing assertions when functions execute
<code>complex.h</code>	Computing common complex mathematical functions
<code>ctype.h</code>	Classifying characters
<code>errno.h</code>	Testing error codes reported by library functions
<code>fenv.h</code>	Floating-point exception flags
<code>float.h</code>	Testing floating-point type properties
<code>inttypes.h</code>	Defining formatters for all types defined in <code>stdint.h</code>
<code>iso646.h</code>	Using Amendment 1— <code>iso646.h</code> standard header
<code>limits.h</code>	Testing integer type properties
<code>locale.h</code>	Adapting to different cultural conventions
<code>math.h</code>	Computing common mathematical functions
<code>setjmp.h</code>	Executing non-local goto statements
<code>signal.h</code>	Controlling various exceptional conditions
<code>stdarg.h</code>	Accessing a varying number of arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C.
<code>stddef.h</code>	Defining several useful types and macros
<code>stdint.h</code>	Providing integer characteristics
<code>stdio.h</code>	Performing input and output
<code>stdlib.h</code>	Performing a variety of operations
<code>string.h</code>	Manipulating several kinds of strings
<code>tgmath.h</code>	Type-generic mathematical functions
<code>time.h</code>	Converting between various time and date formats
<code>uchar.h</code>	Unicode functionality (IAR extension to Standard C)
<code>wchar.h</code>	Support for wide characters

Table 43: Traditional Standard C header files—DLIB

Header file	Usage
wctype.h	Classifying wide characters

Table 43: Traditional Standard C header files—DLIB (Continued)

C++ HEADER FILES

This section lists the C++ header files:

- The C++ library header files
The header files that constitute the Embedded C++ library.
- The C++ standard template library (STL) header files
The header files that constitute STL for the Extended Embedded C++ library.
- The C++ C header files
The C++ header files that provide the resources from the C library.

The C++ library header files

This table lists the header files that can be used in Embedded C++:

Header file	Usage
complex	Defining a class that supports complex arithmetic
fstream	Defining several I/O stream classes that manipulate external files
iomanip	Declaring several I/O stream manipulators that take an argument
ios	Defining the class that serves as the base for many I/O streams classes
iosfwd	Declaring several I/O stream classes before they are necessarily defined
iostream	Declaring the I/O stream objects that manipulate the standard streams
istream	Defining the class that performs extractions
new	Declaring several functions that allocate and free storage
ostream	Defining the class that performs insertions
sstream	Defining several I/O stream classes that manipulate string containers
streambuf	Defining classes that buffer I/O stream operations
string	Defining a class that implements a string container
stringstream	Defining several I/O stream classes that manipulate in-memory character sequences

Table 44: C++ header files

The C++ standard template library (STL) header files

The following table lists the standard template library (STL) header files that can be used in Extended Embedded C++:

Header file	Description
<code>algorithm</code>	Defines several common operations on sequences
<code>deque</code>	A deque sequence container
<code>functional</code>	Defines several function objects
<code>hash_map</code>	A map associative container, based on a hash algorithm
<code>hash_set</code>	A set associative container, based on a hash algorithm
<code>iterator</code>	Defines common iterators, and operations on iterators
<code>list</code>	A doubly-linked list sequence container
<code>map</code>	A map associative container
<code>memory</code>	Defines facilities for managing memory
<code>numeric</code>	Performs generalized numeric operations on sequences
<code>queue</code>	A queue sequence container
<code>set</code>	A set associative container
<code>slist</code>	A singly-linked list sequence container
<code>stack</code>	A stack sequence container
<code>utility</code>	Defines several utility components
<code>vector</code>	A vector sequence container

Table 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:

Header file	Usage
<code>cassert</code>	Enforcing assertions when functions execute
<code>cctype</code>	Classifying characters
<code>cerrno</code>	Testing error codes reported by library functions
<code>cfloat</code>	Testing floating-point type properties
<code>cinttypes</code>	Defining formatters for all types defined in <code>stdint.h</code>

Table 46: New Standard C header files—DLIB

Header file	Usage
<code>climits</code>	Testing integer type properties
<code>locale</code>	Adapting to different cultural conventions
<code>cmath</code>	Computing common mathematical functions
<code>csetjmp</code>	Executing non-local goto statements
<code>csignal</code>	Controlling various exceptional conditions
<code>cstdarg</code>	Accessing a varying number of arguments
<code>cstdbool</code>	Adds support for the <code>bool</code> data type in C.
<code>cstddef</code>	Defining several useful types and macros
<code>cstdint</code>	Providing integer characteristics
<code>stdio</code>	Performing input and output
<code>stdlib</code>	Performing a variety of operations
<code>string</code>	Manipulating several kinds of strings
<code>ctime</code>	Converting between various time and date formats
<code>wchar</code>	Support for wide characters
<code>wctype</code>	Classifying wide characters

Table 46: New Standard C header files—DLIB (Continued)

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

ADDED C FUNCTIONALITY

The DLIB runtime environment includes some added C functionality.

The following include files provide these features:

- `fenv.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

fenv.h

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapeenable` and `fegettrapdisable`.

stdio.h

These functions provide additional I/O functionality:

<code>fdopen</code>	Opens a file based on a low-level file descriptor.
<code>fileno</code>	Gets the low-level file descriptor from the file descriptor (<code>FILE*</code>).
<code>__gets</code>	Corresponds to <code>fgets</code> on <code>stdin</code> .
<code>getw</code>	Gets a <code>wchar_t</code> character from <code>stdin</code> .
<code>putw</code>	Puts a <code>wchar_t</code> character to <code>stdout</code> .
<code>__ungetchar</code>	Corresponds to <code>ungetc</code> on <code>stdout</code> .
<code>__write_array</code>	Corresponds to <code>fwrite</code> on <code>stdout</code> .

string.h

These are the additional functions defined in `string.h`:

<code>strdup</code>	Duplicates a string on the heap.
<code>strcasemp</code>	Compares strings case-insensitive.
<code>strncasemp</code>	Compares strings case-insensitive and bounded.
<code>strlen</code>	Bounded string length.

time.h

There are two interfaces for using `time_t` and the associated functions `time`, `ctime`, `difftime`, `gmtime`, `localtime`, and `mktime`:

- The 32-bit interface supports years from 1900 up to 2035 and uses a 32-bit integer for `time_t`. The type and function have names like `__time32_t`, `__time32`, etc. This variant is mainly available for backwards compatibility.
- The 64-bit interface supports years from -9999 up to 9999 and uses a signed `long long` for `time_t`. The type and function have names like `__time64_t`, `__time64`, etc.

In both interfaces, `time_t` starts at the year 1970.

The interfaces are defined in the system header file `time.h`.

An application can use either interface, and even mix them by explicitly using the 32- or 64-bit variants. By default, the library and the header redirect `time_t`, `time` etc. to

the 32-bit variants. However, to explicitly redirect them to their 64-bit variants, define `_DLIB_TIME_USES_64` in front of the inclusion of `time.h` or `ctime`.

See also, `__time32`, `__time64`, page 145.

`clock_t` is represented by a 32-bit integer type.

SYMBOLS USED INTERNALLY BY THE LIBRARY

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__code`

This symbol is used as a memory attribute internally by the compiler, and might have to be used as an argument in certain templates.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor`, `__has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

Note: The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

CLIB runtime environment—implementation details

The CLIB runtime environment provides most of the important C standard library definitions that apply to embedded systems. These are of the following types:

- Standard C library definitions available for user programs. These are documented in this chapter.
- The system startup code; see the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of MSP430 features. See the chapter *Intrinsic functions* for more information.

LIBRARY DEFINITIONS SUMMARY

This table lists the C header files specific to the CLIB runtime environment:

Header file	Description
<code>assert.h</code>	Assertions
<code>ctype.h*</code>	Character handling
<code>errno.h</code>	Error return values
<code>float.h</code>	Limits and sizes of floating-point types
<code>iccbutl.h</code>	Low-level routines
<code>limits.h</code>	Limits and sizes of integral types
<code>math.h</code>	Mathematics
<code>setjmp.h</code>	Non-local jumps
<code>stdarg.h</code>	Variable arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C
<code>stddef.h</code>	Common definitions including <code>size_t</code> , <code>NULL</code> , <code>ptrdiff_t</code> , and <code>offsetof</code>
<code>stdio.h</code>	Input/output
<code>stdlib.h</code>	General utilities
<code>string.h</code>	String handling

Table 47: CLIB runtime environment header files

* The functions `isxxx`, `toupper`, and `tolower` declared in the header file `ctype.h` evaluate their argument more than once. This is not according to the ISO/ANSI standard.

Segment reference

- Summary of segments
- Descriptions of segments

For more information about placement of segments, see the chapter *Linking your application*.

Summary of segments

The table below lists the segments that are available in the compiler:

Segment	Description
BSLSIGNATURE	Holds BSL configuration data recognized by the device.
CHECKSUM	Holds the checksum generated by the linker.
CODE	Holds program code.
CODE_I	Holds code declared <code>__ramfunc</code> .
CODE_ID	Holds code copied to <code>CODE_I</code> at startup.
CODE_PAD	Holds padding NOP instructions between the <code>CODE</code> and <code>MPU_B2</code> segments.
CODE16	Holds program code in the Small code model.
CSTACK	Holds the stack used by C or C++ programs.
CSTART	Holds the startup code.
DATA16_AC	Holds <code>__data16</code> located constant data.
DATA16_AN	Holds <code>__data16</code> located uninitialized data.
DATA16_C	Holds <code>__data16</code> constant data.
DATA16_HEAP	Holds the heap used for dynamically allocated data in <code>data16</code> memory.
DATA16_I	Holds <code>__data16</code> static and global initialized variables.
DATA16_ID	Holds initial values for <code>__data16</code> static and global variables in <code>DATA16_I</code> .
DATA16_N	Holds <code>__no_init</code> <code>__data16</code> static and global variables.
DATA16_P	Holds <code>__data16</code> variables defined with the <code>__persistent</code> keyword.
DATA16_Z	Holds zero-initialized <code>__data16</code> static and global variables.

Table 48: Segment summary

Segment	Description
DATA20_AC	Holds <code>__data20</code> located constant data.
DATA20_AN	Holds <code>__data20</code> located uninitialized data.
DATA20_C	Holds <code>__data20</code> constant data.
DATA20_HEAP	Holds the heap used for dynamically allocated data in <code>data20</code> memory.
DATA20_I	Holds <code>__data20</code> static and global initialized variables.
DATA20_ID	Holds initial values for <code>__data20</code> static and global variables in <code>DATA20_I</code> .
DATA20_N	Holds <code>__no_init __data20</code> static and global variables.
DATA20_P	Holds <code>__data20</code> variables defined with the <code>__persistent</code> keyword.
DATA20_Z	Holds zero-initialized <code>__data20</code> static and global variables.
DIFUNCT	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before <code>main</code> is called.
INFO	Holds data to be placed in the MSP430 information memory.
INFOA	Holds data to be placed in bank A of the MSP430 information memory.
INFOB	Holds data to be placed in bank B of the MSP430 information memory.
INFOC	Holds data to be placed in bank C of the MSP430 information memory.
INFOD	Holds data to be placed in bank D of the MSP430 information memory.
INTVEC	Holds the interrupt vector.
IPE_B1	Defines the start of the IPE address range.
IPE_B2	Defines the end of the IPE address range.
IPECODE16	Holds code inside the IPE address range.
IPEDATA16_C	Holds constant data inside the IPE address range.
IPEDATA16_N	Holds writable <code>__no_init</code> data inside the IPE address range.
IPESIGNATURE	Holds IPE configuration data recognized by the device.
ISR_CODE	Holds interrupt functions when compiling for the MSP430X architecture.
JTAGSIGNATURE	Holds JTAG configuration data recognized by the device.
MPU_B1	Defines the border between the first and second MPU ranges.
MPU_B2	Defines the border between the second and third MPU ranges.
REGVAR_AN	Holds <code>__regvar</code> data.
RESET	Holds the reset vector.
SIGNATURE	Holds configuration data recognized by the device. This segment is deprecated but kept for backward compatibility.

Table 48: Segment summary (Continued)

Segment	Description
TLS16_I	Holds thread-local static and global initialized variables used by the main thread.
TLS16_ID	Holds initial values for thread-local static and global variables in TLS16_I.

Table 48: Segment summary (Continued)

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives `-z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous. For information about these directives, see *Using the -Z command for sequential placement*, page 103 and *Using the -P command for packed placement*, page 103, respectively.

For each segment, the segment memory type is specified, which indicates in which type of memory the segment should be placed; see *Segment memory type*, page 86.

For information about how to define segments in the linker configuration file, see *Linking your application*, page 101.

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

BSLSIGNATURE

Description	Holds BSL configuration data recognized by the device.
Segment memory type	CONST
Memory placement	Device-specific, typically 0xFF84–0xFF87.
Access type	Read-only

CHECKSUM

Description	Holds the checksum bytes generated by the linker. This segment also holds the <code>__checksum</code> symbol. Note that the size of this segment is affected by the linker option <code>-J</code> .
-------------	---

Segment memory type

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, unless the Small code model is used (MSP430X only). For MSP430, this segment also holds interrupt code.

Segment memory type CODE

Memory placement MSP430: 0x0002-0xFFFFD
MSP430X: 0x00002-0xFFFFFE

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: 0x0002-0xFFFFD
MSP430X: 0x00002-0xFFFFFE

Access type Read-write

CODE_ID

Description 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.

Segment memory type CODE

Memory placement MSP430: 0x0002-0xFFFFD
MSP430X: 0x00002-0xFFFFFD, 0x10040-0xFFFFD

Access type	Read-only
-------------	-----------

CODE_PAD

Description	Holds padding NOP instructions between the CODE and MPU_B2 segments, to prevent the instruction prefetch mechanism from violating MPU access restrictions.
-------------	--

Segment memory type	CODE
---------------------	------

Memory placement	0x0-0xFFFFF
------------------	-------------

Access type	Read-only
-------------	-----------

CODE16

Description	Holds program code, except the code for system initialization, in the Small code model (MSP430X only).
-------------	--

Segment memory type	CODE
---------------------	------

Memory placement	0x00002-0x0FFFD
------------------	-----------------

Access type	Read-only
-------------	-----------

CSTACK

Description	Holds the internal data stack.
-------------	--------------------------------

Segment memory type	DATA
---------------------	------

Memory placement	0x0002-0xFFFFD (also for the MSP430X architecture)
------------------	--

Access type	Read-write
-------------	------------

See also	<i>Setting up stack memory</i> , page 105.
----------	--

CSTART

Description	<p>Holds the startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-z</code> directive must be used.</p>
Segment memory type	CODE
Memory placement	0x0002-0xFFFFD
Access type	Read-only

DATA16_AC

Description	<p>Holds <code>__data16</code> located constant data.</p> <p><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file.</p>
-------------	--

DATA16_AN

Description	<p>Holds <code>__no_init __data16</code> located data.</p> <p><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file.</p>
-------------	---

DATA16_C

Description	<p>Holds <code>__data16</code> constant data. This can include constant variables, string and aggregate literals, etc.</p>
Segment memory type	CONST
Memory placement	0x0001-0xFFFFD
Access type	Read-only

DATA16_HEAP

Description	Holds the heap used for dynamically allocated data in data16 memory, in other words data allocated by <code>data16_malloc</code> and <code>data16_free</code> , and in C++, <code>new</code> and <code>delete</code> .
Segment memory type	DATA
Memory placement	0x0002-0xFFFFD
Access type	Read-write
See also	<i>Setting up heap memory</i> , page 106 and <i>New and Delete operators</i> , page 207.

DATA16_I

Description	Holds <code>__data16</code> static and global initialized variables initialized by copying from the segment <code>DATA16_ID</code> at application startup. This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-z</code> directive must be used.
Segment memory type	DATA
Memory placement	0x0001-0xFFFFD
Access type	Read-write

DATA16_ID

Description	Holds initial values for <code>__data16</code> static and global variables in the <code>DATA16_I</code> segment. These values are copied from <code>DATA16_ID</code> to <code>DATA16_I</code> at application startup. This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-z</code> directive must be used.
Segment memory type	CONST
Memory placement	0x0001-0xFFFFD
Access type	Read-only

DATA16_N

Description	Holds static and global <code>__no_init __data16</code> variables.
Segment memory type	DATA
Memory placement	0x0001-0xFFFFD
Access type	Read-write

DATA16_P

Description	Holds static and global <code>__data16</code> variables defined with the <code>__persistent</code> keyword. These variables will only be initialized, for example, by a code downloader, and not by <code>cstartup</code> .
Segment memory type	CONST
Memory placement	0x0001-0xFFFFD
Access type	Read-write
See also	

DATA16_Z

Description	<p>Holds zero-initialized <code>__data16</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-z</code> directive must be used.</p>
Segment memory type	DATA
Memory placement	0x0001-0xFFFFD
Access type	Read-write

DATA20_AC

Description	<p>Holds <code>__data20</code> located constant data.</p> <p><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file.</p> <p>Note: The compiler requires that no data is placed in the address range <code>0x10000-0x1003F</code>.</p>
-------------	--

DATA20_AN

Description	<p>Holds <code>__no_init __data20</code> located data.</p> <p><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file.</p> <p>Note: The compiler requires that no data is placed in the address range <code>0x10000-0x1003F</code>.</p>
-------------	---

DATA20_C

Description	<p>Holds <code>__data20</code> constant data. This can include constant variables, string and aggregate literals, etc.</p> <p>This segment is only available if you are using MSP430X.</p>
Segment memory type	CONST
Memory placement	0x00001-0x0FFFE, 0x10040-0xFFFFE
Access type	Read-only

DATA20_HEAP

Description	<p>Holds the heap used for dynamically allocated data in <code>data20</code> memory, in other words data allocated by <code>data20_malloc</code> and <code>data20_free</code>, and in C++, <code>new</code> and <code>delete</code>.</p> <p>This segment is only available if you are using MSP430X.</p>
Segment memory type	DATA

Memory placement	0x00001-0x0FFFE, 0x10040-0xFFFFE
Access type	Read-write
See also	<i>Setting up heap memory</i> , page 106 and <i>New and Delete operators</i> , page 207.

DATA20_I

Description	<p>Holds <code>__data20</code> static and global initialized variables initialized by copying from the segment <code>DATA20_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-z</code> directive must be used.</p> <p>This segment is only available if you are using MSP430X.</p>
Segment memory type	DATA
Memory placement	0x00001-0x0FFFE, 0x10040-0xFFFFE
Access type	Read-write

DATA20_ID

Description	<p>Holds initial values for <code>__data20</code> static and global variables in the <code>DATA20_I</code> segment. These values are copied from <code>DATA20_ID</code> to <code>DATA20_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-z</code> directive must be used.</p> <p>This segment is only available if you are using MSP430X.</p>
Segment memory type	CONST
Memory placement	0x00001-0x0FFFE, 0x10040-0xFFFFE
Access type	Read-only

DATA20_N

Description	Holds static and global <code>__no_init __data20</code> variables. This segment is only available if you are using MSP430X.
Segment memory type	DATA
Memory placement	0x00001-0x0FFFE, 0x10040-0xFFFFE
Access type	Read-write

DATA20_P

Description	Holds static and global <code>__data20</code> variables defined with the <code>__persistent</code> keyword. These variables will only be initialized, for example, by a code downloader, and not by <code>cstartup</code> . This segment is only available if you are using MSP430X.
Segment memory type	CONST
Memory placement	0x00001-0x0FFFE, 0x10040-0xFFFFE
Access type	Read-write
See also	<code>__persistent</code> , page 318

DATA20_Z

Description	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. This segment is only available if you are using MSP430X.
Segment memory type	DATA
Memory placement	0x00001-0x0FFFE, 0x10040-0xFFFFE
Access type	Read-write

DIFUNCT

Description	Holds the dynamic initialization vector used by C++.
Segment memory type	CONST
Memory placement	This segment must be placed in the first 64 Kbytes of memory.
Access type	Read-only

INFO

Description	Holds data to be placed in the MSP430 information memory. Note that the <code>INFO</code> segment and the <code>INFOA-INFOD</code> segments overlap.
Segment memory type	CONST
Memory placement	Depends on the device.
Access type	Read-only

INFOA

Description	Holds data to be placed in bank A of the MSP430 information memory. Note that the <code>INFOA</code> segment and the <code>INFO</code> 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 <code>INFOB</code> segment and the <code>INFO</code> 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.
------------------	------------------------

Access type	Read-only
-------------	-----------

INTVEC

Description	Holds the interrupt vector table generated by the use of the <code>__interrupt</code> extended keyword in combination with the <code>#pragma vector</code> directive.
-------------	---

Segment memory type	CODE
---------------------	------

Memory placement	This segment can be placed anywhere in memory.
------------------	--

Access type	Read-only
-------------	-----------

IPE_BI

Description	The start of this segment defines the start of the protected IPE address range. It holds the IPE control <code>struct</code> which is read by the boot code to initialize the IPE.
-------------	--

Segment memory type	CONST
Memory placement	0x0-0xFFFF
Access type	Read-only

IPE_B2

Description	The start of this segment defines the end of the protected IPE address range. It only contains a zero-size code fragment that constrains the alignment of the IPE end address.
Segment memory type	CONST
Memory placement	0x0-0xFFFF
Access type	Not applicable

IPECODEI6

Description	Holds code inside the protected IPE address range. Code in this segment can be used for reading data located inside the protected IPE address range.
Segment memory type	CODE
Memory placement	0x0-0xFFFF, between IPE_B1 and IPE_B2
Access type	Read-only

IPEDATAI6_C

Description	Holds constants inside the protected IPE address range. Data in this segment can only be read by code located inside the protected IPE address range.
Segment memory type	CONST
Memory placement	0x0-0xFFFF, between IPE_B1 and IPE_B2
Access type	Read-only

IPEDATAI6_N

Description	Holds writable <code>__no_init</code> data inside the protected IPE address range. Data inside this range can only be read or written by code located inside the protected IPE address range.
Segment memory type	DATA
Memory placement	0x0-0xFFFF, between IPE_B1 and IPE_B2.
Access type	Read-write

IPESIGNATURE

Description	Holds IPE configuration data recognized by the device.
Segment memory type	CONST
Memory placement	Device-specific, typically 0xFF88-0xFF8F.
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	0x0002-0xFFFFD
Access type	Read-only

JTAGSIGNATURE

Description	Holds JTAG configuration data recognized by the device.
Segment memory type	CONST
Memory placement	Device-specific, typically 0xFF80-0xFF83.
Access type	Read-only

MPU_B1

Description	The <code>__iar_430_mpu_init</code> function uses the start of this segment to define the border between the first and second MPU address ranges. This segment only contains a zero-size code fragment that constrains the alignment of the MPU border.
Segment memory type	CONST
Memory placement	0x0-0xFFFFF
Access type	Not applicable.

MPU_B2

Description	The <code>__iar_430_mpu_init</code> function uses the start of this segment to define the border between the second and third MPU address ranges. This segment only contains a zero-size code fragment that constrains the alignment of the MPU border.
Segment memory type	CONST
Memory placement	0x0-0xFFFFF
Access type	Not applicable.

REGVAR_AN

Description	Holds <code>__regvar</code> 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-0xFFFF
Access type	Read-only

SIGNATURE

Description	Holds configuration data recognized by the device, such as the JTAG password and the IPE signature. Note: This segment is deprecated, but kept for backward compatibility.
Segment memory type	CONST
Memory placement	Device-specific, typically 0xFF80–0xFF8F
Access type	Read-only

TLS16_I

Description	Holds thread-local static and global initialized variables used by the main thread, initialized by copying from the segment <code>TLS16_ID</code> at application startup. This segment is only used by custom-built runtime libraries configured with thread support. This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the content must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.
Segment memory type	DATA
Memory placement	0x0001–0xFFFFD
Access type	Read-write

TLS16_ID

Description	Holds initial values for thread-local static and global variables in the <code>TLS16_I</code> segment for the main thread, and for each started thread. The main thread is initialized by the startup code; all other threads are initialized by the embedded operating system. This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the content must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.
Segment memory type	CONST
Memory placement	0x0001–0xFFFFD

Access type

Read-only

The stack usage control file

- Overview
- Stack usage control directives
- Syntactic components

Before you read this chapter, see *Stack usage analysis*, page 91.

Overview

A stack usage control file consists of a sequence of directives that control stack usage analysis. You can use C ("*/*...*/*") and C++ ("*//...*") comments in these files.

The default filename extension for stack usage control files is `suc`.

C++ NAMES

You can also use wildcards in function names. "*##**" matches any sequence of characters, and "*#?*" matches a single character.

Stack usage control directives

This section gives detailed reference information about each stack usage control directive.

call graph root directive

Syntax	<code>call graph root [<i>category</i>] : <i>func-spec</i> [, <i>func-spec</i>...] ;</code>				
Parameters	<table><tr><td><i>category</i></td><td>See <i>category</i>, page 401</td></tr><tr><td><i>func-spec</i></td><td>See <i>func-spec</i>, page 401</td></tr></table>	<i>category</i>	See <i>category</i> , page 401	<i>func-spec</i>	See <i>func-spec</i> , page 401
<i>category</i>	See <i>category</i> , page 401				
<i>func-spec</i>	See <i>func-spec</i> , page 401				
Description	Specifies that the listed functions are call graph roots. You can optionally specify a call graph root category. Call graph roots are listed under their category in the <i>Stack Usage</i> chapter in the linker map file.				

The linker will normally issue a warning for functions needed in the application that are not call graph roots and which do not appear to be called.

Example `call graph root [task]: MyFunc10, MyFunc11;`

See also *call_graph_root*, page 327.

check that directive

Syntax `check that expression;`

Parameters

<i>expression</i>	A boolean expression.
-------------------	-----------------------

Description You can use the `check that` directive to compare the results of stack usage analysis against the sizes of blocks and regions. If the expression evaluates to zero, an error is emitted.

Three extra operators are available for use only in `check that` expressions:

<code>maxstack(<i>category</i>)</code>	The stack depth of the deepest call chain for any call graph root function in the category.
<code>totalstack(<i>category</i>)</code>	The sum of the stack depths of the deepest call chains for each call graph root function in the category.
<code>size("SEGMENT")</code>	The size of the segment.

Example

```
check that maxstack("Program entry")
           + totalstack("interrupt")
           + 1K
           <= size("CSTACK");
```

See also *Stack usage analysis*, page 91.

exclude directive

Syntax `exclude func-spec [, func-spec...];`

Parameters

<i>func-spec</i>	See <i>func-spec</i> , page 401
------------------	---------------------------------

Description	Excludes the specified functions, and call trees originating with them, from stack usage calculations.
Example	<code>exclude MyFunc5, MyFunc6;</code>

function directive

Syntax	<code>[<i>override</i>] function [<i>category</i>] <i>func-spec</i> : <i>stack-size</i> [, <i>call-info...</i>];</code>								
Parameters	<table> <tr> <td><i>category</i></td> <td>See category, page 401</td> </tr> <tr> <td><i>func-spec</i></td> <td>See func-spec, page 401</td> </tr> <tr> <td><i>call-info</i></td> <td>See call-info, page 402</td> </tr> <tr> <td><i>stack-size</i></td> <td>See stack-size, page 403</td> </tr> </table>	<i>category</i>	See category, page 401	<i>func-spec</i>	See func-spec, page 401	<i>call-info</i>	See call-info, page 402	<i>stack-size</i>	See stack-size, page 403
<i>category</i>	See category, page 401								
<i>func-spec</i>	See func-spec, page 401								
<i>call-info</i>	See call-info, page 402								
<i>stack-size</i>	See stack-size, page 403								
Description	<p>Specifies what the maximum stack usage is in a function and which other functions that are called from that function.</p> <p>Normally, an error is issued if there already is stack usage information for the function, but if you start with <code>override</code>, the error will be suppressed and the information supplied in the directive will be used instead of the previous information.</p>								
Example	<pre>function MyFunc1: 32, calls MyFunc2, calls MyFunc3, MyFunc4: 16; function [interrupt] MyInterruptHandler: 44;</pre>								

max recursion depth directive

Syntax	<code>max recursion depth <i>func-spec</i> : <i>size</i>;</code>				
Parameters	<table> <tr> <td><i>func-spec</i></td> <td>See func-spec, page 401</td> </tr> <tr> <td><i>size</i></td> <td>See size, page 403</td> </tr> </table>	<i>func-spec</i>	See func-spec, page 401	<i>size</i>	See size, page 403
<i>func-spec</i>	See func-spec, page 401				
<i>size</i>	See size, page 403				
Description	Specifies the maximum number of iterations through any of the cycles in the recursion nest of which the function is a member.				

A recursion nest is a set of cycles in the call graph where each cycle shares at least one node with another cycle in the nest.

Stack usage analysis will base its result on the max recursion depth multiplied by the stack usage of the deepest cycle in the nest. If the nest is not entered on a point along one of the deepest cycles, no stack usage result will be calculated for such calls.

Example

```
max recursion depth MyFunc12: 10;
```

no calls from directive**Syntax**

```
no calls from module-spec to func-spec [ , func-spec... ];
```

Parameters

func-spec

See *func-spec*, page 401

module-spec

See *module-spec*, page 402

Description

When you provide stack usage information for some functions in a module without stack usage information, the linker warns about functions that are referenced from the module but not listed as called. This is primarily to help avoid problems with C runtime routines, calls to which are generated by the compiler, beyond user control.

If there actually is no call to some of these functions, use the `no calls from` directive to selectively suppress the warning for the specified functions. You can also disable the warning entirely (`--diag_suppress` or

Project>Options>Linker>Diagnostics>Suppress these diagnostics).

Example

```
no calls from [file.r43] to MyFunc13, MyFun14;
```

possible calls directive**Syntax**

```
possible calls calling-func : called-func [ , called-func... ];
```

Parameters

calling-func

See *func-spec*, page 401

called-func

See *func-spec*, page 401

Description

Specifies an exhaustive list of possible destinations for all indirect calls in one function. Use this for functions which are known to perform indirect calls and where you know exactly which functions that might be called in this particular application. Consider

	using the <code>#pragma calls</code> directive if the information about which functions that might be called is available when compiling.
Example	<pre>possible calls MyFunc7: MyFunc8, MyFunc9;</pre> <p>When the function does not perform any calls, the list is empty:</p> <pre>possible calls MyFunc8: ;</pre>
See also	<i>calls</i> , page 327.

Syntactic components

This section describes the syntactical components that can be used by the stack usage control directives.

category

Syntax	<code>[name]</code>
Description	A call graph root category. You can use any name you like. Categories are not case-sensitive.
Example	<p>category examples:</p> <pre>[interrupt] [task]</pre>

func-spec

Syntax	<code>[?] name [module-spec]</code>
Description	Specifies the name of a symbol, and for module-local symbols, the name of the module it is defined in. Normally, if <i>func-spec</i> does not match a symbol in the program, a warning is emitted. Prefixing with <code>?</code> suppresses this warning.
Example	<p><i>func-spec</i> examples:</p> <pre>xFun MyFun [file.r43]</pre>

module-spec

Syntax	<code>[name [(name)]]</code>
Description	<p>Specifies the name of a module, and optionally, in parentheses, the name of the library it belongs to. To distinguish between modules with the same name, you can specify:</p> <ul style="list-style-type: none"> • The complete path of the file ("D:\C1\test\file.o") • As many path elements as are needed at the end of the path ("test\file.o") • Some path elements at the start of the path, followed by "...", followed by some path elements at the end ("D:\...\file.o"). <p>Note that when using multi-file compilation (<code>--mfc</code>), multiple files are compiled into a single module, named after the first file.</p>
Example	<p><i>module-spec</i> examples:</p> <pre> [file.r43] [file.r43(lib.a)] ["D:\C1\test\file.r43"] </pre>

name

Description	<p>A name can be either an identifier or a quoted string.</p> <p>The first character of an identifier must be either a letter or one of the characters "_", "\$", or ".". The rest of the characters can also be digits.</p> <p>A quoted string starts and ends with " and can contain any character. Two consecutive " characters can be used inside a quoted string to represent a single ".</p>
Example	<p><i>name</i> examples:</p> <pre> MyFun file.r43 "file-1.r43" </pre>

call-info

Syntax	<code>calls func-spec [, func-spec...] [: stack-size]</code>
Description	Specifies one or more called functions, and optionally, the stack size at the calls.

Example `call-info` examples:

```
calls MyFunc1 : stack 16
calls MyFunc2, MyFunc3, MyFunc4
```

stack-size

Syntax `[stack] size`

Description Specifies the size of a stack frame.

Example `stack-size` examples:

```
24
stack 28
```

size

Description A decimal integer, or 0x followed by a hexadecimal integer. Either alternative can optionally be followed by a suffix indicating a power of two ($K=2^{10}$, $M=2^{20}$, $G=2^{30}$, $T=2^{40}$, $P=2^{50}$).

Example `size` examples:

```
24
0x18
2048
2K
```


Implementation-defined behavior for Standard C

- Descriptions of implementation-defined behavior

If you are using C89 instead of Standard C, see Implementation-defined behavior for C89, page 421. For a short overview of the differences between Standard C and C89, see *C language overview*, page 193.

The text in this chapter applies to the DLIB runtime environment. Because the CLIB runtime environment does not follow Standard C, its implementation-defined behavior is not documented. See also *The CLIB runtime environment*, page 159.

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.

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.

J.3.1 TRANSLATION

Diagnostics (3.10, 5.1.1.3)

Diagnostics are produced in the form:

filename, *linenumber* *level*[*tag*]: *message*

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.

J.3.2 ENVIRONMENT**The character set (5.1.1.2)**

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, if you use the `--enable_multibytes` compiler option, the host character set is used instead.

Main (5.1.2.1)

The function called at program startup is called `main`. No prototype is declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the DLIB runtime environment, see *Customizing `__low_level_init`*, page 138.

The effect of program termination (5.1.2.1)

Terminating the application returns the execution to the startup code (just after the call to `main`).

Alternative ways to define main (5.1.2.2.1)

There is no alternative ways to define the `main` function.

The argv argument to main (5.1.2.2.1)

The `argv` argument is not supported.

Streams as interactive devices (5.1.2.3)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

Signals, their semantics, and the default handling (7.14)

In the DLIB runtime environment, 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 runtime environment, there are no implementation-defined values that correspond to a computational exception.

Signals at system startup (7.14.1.1)

In the DLIB runtime environment, there are no implementation-defined signals that are executed at system startup.

Environment names (7.20.4.5)

In the DLIB runtime environment, there are no implementation-defined environment names that are used by the `getenv` function.

The system function (7.20.4.6)

The `system` function is not supported.

J.3.3 IDENTIFIERS

Multibyte characters in identifiers (6.4.2)

Additional multibyte characters may not appear in identifiers.

Significant characters in identifiers (5.2.4.1, 6.1.2)

The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

J.3.4 CHARACTERS

Number of bits in a byte (3.6)

A byte contains 8 bits.

Execution character set member values (5.2.1)

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the host character set.

Alphabetic escape sequences (5.2.2)

The standard alphabetic escape sequences have the values `\a-7`, `\b-8`, `\f-12`, `\n-10`, `\r-13`, `\t-9`, and `\v-11`.

Characters outside of the basic executive character set (6.2.5)

A character outside of the basic executive character set that is stored in a `char` is not transformed.

Plain char (6.2.5, 6.3.1.1)

A plain `char` is treated as an `unsigned char`.

Source and execution character sets (6.4.4.4, 5.1.1.2)

The source character set is the set of legal characters that can appear in source files. By default, the source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. By default, the execution character set is the standard ASCII character set.

However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 147.

Integer character constants with more than one character (6.4.4.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

Wide character constants with more than one character (6.4.4.4)

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Locale used for wide character constants (6.4.4.4)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Locale used for wide string literals (6.4.5)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Source characters as executive characters (6.4.5)

All source characters can be represented as executive characters.

J.3.5 INTEGERS**Extended integer types (6.2.5)**

There are no extended integer types.

Range of integer values (6.2.6.2)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

For information about the ranges for the different integer types, see *Basic data types—integer types*, page 296.

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, except for the operator `>>` which will behave as an arithmetic right shift.

J.3.6 FLOATING POINT**Accuracy of floating-point operations (5.2.4.2.2)**

The accuracy of floating-point operations is unknown.

Rounding behaviors (5.2.4.2.2)

There are no non-standard values of `FLT_ROUNDS`.

Evaluation methods (5.2.4.2.2)

There are no non-standard values of `FLT_EVAL_METHOD`.

Converting integer values to floating-point values (6.3.1.4)

When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Converting floating-point values to floating-point values (6.3.1.5)

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Denoting the value of floating-point constants (6.4.4.2)

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Contraction of floating-point values (6.5)

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

Default state of `FENV_ACCESS` (7.6.1)

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

Additional floating-point mechanisms (7.6, 7.12)

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

Default state of `FP_CONTRACT` (7.12.2)

The default state of the pragma directive `FP_CONTRACT` is `OFF`.

J.3.7 ARRAYS AND POINTERS

Conversion from/to pointers (6.3.2.3)

For information about casting of data pointers and function pointers, see *Casting*, page 302.

`ptrdiff_t` (6.5.6)

For information about `ptrdiff_t`, see *ptrdiff_t*, page 302.

J.3.8 HINTS

Honoring the register keyword (6.7.1)

User requests for register variables are not honored.

Inlining functions (6.7.4)

User requests for inlining functions increases the chance, but does not make it certain, that the function will actually be inlined into another function. See *Inlining functions*, page 81.

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 298.

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 270.

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 298.

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 295.

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 305.

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 361.

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 262.

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 250.

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 250.

Preprocessing tokens in #include directives (6.10.2)

Preprocessing tokens in an `#include` directive are combined in the same way as outside an `#include` directive.

Nesting limits for #include directives (6.10.2)

There is no explicit nesting limit for `#include` processing.

Universal character names (6.10.3.2)

Universal character names (UCN) are not supported.

Recognized pragma directives (6.10.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the chapter *Pragma directives* and here, the information provided in the chapter *Pragma directives* overrides the information here.

alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_default_requirements
library_provides
library_requirement_override
memory
module_name
no_pch
once
system_include
warnings

Default `__DATE__` and `__TIME__` (6.10.8)

The definitions for `__TIME__` and `__DATE__` are always available.

J.3.12 LIBRARY FUNCTIONS**Additional library facilities (5.1.2.1)**

Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 115.

Diagnostic printed by the `assert` function (7.2.1.1)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

Representation of the floating-point status flags (7.6.2.2)

For information about the floating-point status flags, see *fcntl.h*, page 375.

`feraiseexcept` raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 299.

Strings passed to the `setlocale` function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 147.

Types defined for `float_t` and `double_t` (7.12)

The `FLT_EVAL_METHOD` macro can only have the value 0.

Domain errors (7.12.1)

No function generates other domain errors than what the standard requires.

Return values on domain errors (7.12.1)

Mathematic functions return a floating-point NaN (not a number) for domain errors.

Underflow errors (7.12.1)

Mathematic functions set `errno` to the macro `ERANGE` (a macro in `errno.h`) and return zero for underflow errors.

fmod return value (7.12.10.1)

The `fmod` function returns a floating-point NaN when the second argument is zero.

The magnitude of remquo (7.12.10.3)

The magnitude is congruent modulo `INT_MAX`.

signal() (7.14.1.1)

The signal part of the library is not supported.

Note: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 144 and *raise*, page 143, respectively.

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 *Briefly about input and output (I/O)*, page 116.

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 *Briefly about input and output (I/O)*, page 116.

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 *Briefly about input and output (I/O)*, page 116.

Removal of open temporary files (7.19.4.3)

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

Mode changing (7.19.5.4)

`freopen` closes the named stream, then reopens it in the new mode. The streams `stdin`, `stdout`, and `stderr` can be reopened in any new mode.

Style for printing infinity or NaN (7.19.6.1, 7.24.2.1)

The style used for printing infinity or NaN for a floating-point constant is `inf` and `nan` (`INF` and `NAN` for the `F` conversion specifier), respectively. The `n-char-sequence` is not used for `nan`.

%p in printf() (7.19.6.1, 7.24.2.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

Reading ranges in scanf (7.19.6.2, 7.24.2.1)

A - (dash) character is always treated as a range symbol.

%p in scanf (7.19.6.2, 7.24.2.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

File position errors (7.19.9.1, 7.19.9.3, 7.19.9.4)

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)

An n-char-sequence after a NaN is read and ignored.

errno value at underflow (7.20.1.3, 7.24.4.1.1)

`errno` is set to `ERANGE` if an underflow is encountered.

Zero-sized heap objects (7.20.3)

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

Behavior of abort and exit (7.20.4.1, 7.20.4.4)

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)

The termination status will be propagated to `__exit()` as a parameter. `exit()` and `_Exit()` use the input parameter, whereas `abort` uses `EXIT_FAILURE`.

The system function return value (7.20.4.6)

The `system` function is not supported.

The time zone (7.23.1)

The local time zone and daylight savings time must be defined by the application. For more information, see `__time32`, `__time64`, page 145.

Range and precision of time (7.23)

For information about range and precision, see `time.h`, page 376. The application must supply the actual implementation for the functions `time` and `clock`. See `__time32`, `__time64`, page 145 and `clock`, page 140, respectively.

clock() (7.23.2.1)

The application must supply an implementation of the `clock` function. See `clock`, page 140.

%Z replacement string (7.23.3.5, 7.24.5.1)

By default, "Z" is used as a replacement for %Z. Your application should implement the time zone handling. See `__time32`, `__time64`, page 145.

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 295.

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 295.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 295.

J.4 LOCALE**Members of the source and execution character set (5.2.1)**

By default, the compiler accepts all one-byte characters in the host's default character set. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

The meaning of the additional character set (5.2.1.2)

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

Shift states for encoding multibyte characters (5.2.1.2)

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

Direction of successive printing characters (5.2.2)

The application defines the characteristics of a display device.

The decimal point character (7.1.1)

The default decimal-point character is a '.'. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

Printing characters (7.4, 7.25.2)

The set of printing characters is determined by the chosen locale.

Control characters (7.4, 7.25.2)

The set of control characters is determined by the chosen locale.

Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

The sets of characters tested are determined by the chosen locale.

The native environment (7.1.1.1)

The native environment is the same as the "C" locale.

Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

The collation of the execution character set (7.21.4.3, 7.24.4.4.2)

The collation of the execution character set is determined by the chosen locale.

Message returned by `strerror` (7.21.6.2)

The messages returned by the `strerror` function depending on the argument is:

Argument	Message
EZERO	no error
EDOM	domain error
ERANGE	range error
EFPOS	file positioning error
EILSEQ	multi-byte encoding error
<0 >99	unknown error
all others	error <i>nnn</i>

Table 49: Message returned by `strerror()`—DLIB runtime environment

Implementation-defined behavior for C89

- Descriptions of implementation-defined behavior

If you are using Standard C instead of C89, see *Implementation-defined behavior for Standard C*, page 405. For a short overview of the differences between Standard C and C89, see *C language overview*, page 193.

Descriptions of implementation-defined behavior

The descriptions follow the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

```
filename, linenumber level[tag]: message
```

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the DLIB runtime environment, see *Customizing __low_level_init*, page 138. To change this behavior for the CLIB runtime environment, see *Customizing system initialization*, page 164.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. The CLIB runtime environment does not support multibyte characters.

See *Locale*, page 147.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the

same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the DLIB runtime environment will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The CLIB runtime environment does not support multibyte characters.

See *Locale*, page 147.

Range of 'plain' char (6.2.1.1)

A ‘plain’ char has the same range as an unsigned char.

INTEGERS

Range of integer values (6.1.2.5)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types—integer types*, page 296, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit, except for the operator `>>` which will behave as an arithmetic right shift.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT**Representation of floating-point values (6.1.2.5)**

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (*s*), a biased exponent (*e*), and a mantissa (*m*).

See *Basic data types—floating-point types*, page 299, for information about the ranges and sizes for the different floating-point types: `float` and `double`.

Converting integer values to floating-point values (6.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

Demoting floating-point values (6.2.1.4)

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

ARRAYS AND POINTERS

size_t (6.3.3.4, 7.1.1)

See *size_t*, page 302, for information about *size_t*.

Conversion from/to pointers (6.3.4)

See *Casting*, page 302, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 302, for information about the *ptrdiff_t*.

REGISTERS

Honoring the register keyword (6.5.1)

User requests for register variables are not honored.

STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

Improper access to a union (6.3.2.3)

If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

Padding and alignment of structure members (6.5.2.1)

See the section *Basic data types—integer types*, page 296, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' *int* bitfield is treated as a *signed int* bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source

file currently being processed. If there is no grandparent file and the file is not found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the chapter *Pragma directives* and here, the information provided in the chapter *Pragma directives* overrides the information here.

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_default_requirements
library_provides
```

```
library_requirement_override
memory
module_name
no_pch
once
system_include
warnings
```

Default `__DATE__` and `__TIME__` (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

LIBRARY FUNCTIONS FOR THE IAR DLIB RUNTIME ENVIRONMENT

Note that some items in this list only apply when file descriptors are supported by the library configuration. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to 0.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

Domain errors (7.5.1)

NaN (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

`fmod()` functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns NaN; `errno` is set to `EDOM`.

signal() (7.7.1.1)

The signal part of the library is not supported.

Note: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 144 and *raise*, page 143, respectively.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

No null characters are appended to data written to binary streams.

Files (7.9.3)

Whether the file position indicator of an append-mode stream is initially positioned at the beginning or the end of the file, depends on the application-specific implementation of the low-level file routines.

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 116.

The characteristics of the file buffering is that the implementation supports files that are unbuffered, line buffered, or fully buffered.

Whether a zero-length file actually exists depends on the application-specific implementation of the low-level file routines.

Rules for composing valid file names depends on the application-specific implementation of the low-level file routines.

Whether the same file can be simultaneously open multiple times depends on the application-specific implementation of the low-level file routines.

remove() (7.9.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 116.

rename() (7.9.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 116.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

Message generated by perror() (7.9.10.4)

The generated message is:

usersuppliedprefix:errormessage

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of `exit()` (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *getenv*, page 141.

`system()` (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *system*, page 145.

Message returned by `strerror()` (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

Argument	Message
EZERO	no error
EDOM	domain error
ERANGE	range error
EFPOS	file positioning error
EILSEQ	multi-byte encoding error
<0 >99	unknown error
all others	error <i>nnn</i>

Table 50: Message returned by `strerror()`—DLIB runtime environment

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in `__time32`, `__time64`, page 145.

`clock()` (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *clock*, page 140.

LIBRARY FUNCTIONS FOR THE CLIB RUNTIME ENVIRONMENT

NULL macro (7.1.6)

The NULL macro is defined to `(void *) 0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

Assertion failed: *expression*, file *Filename*, line *linenumber*

when the parameter evaluates to zero.

Domain errors (7.5.1)

`HUGE_VAL`, the largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

`fmod()` functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns zero (it does not change the integer expression `errno`).

`signal()` (7.7.1.1)

The signal part of the library is not supported.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

There are no binary streams implemented.

Files (7.9.3)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

remove() (7.9.4.1)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

rename() (7.9.4.2)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `'void *'`.

Reading ranges in scanf() (7.9.6.2)

A `-` (dash) character is always treated explicitly as a `-` character.

File position errors (7.9.9.1, 7.9.9.4)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

Message generated by perror() (7.9.10.4)

`perror()` is not supported.

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The `exit()` function does not return.

Environment (7.10.4.4)

Environments are not supported.

system() (7.10.4.5)

The `system()` function is not supported.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument are:

Argument	Message
EZERO	no error
EDOM	domain error
ERANGE	range error
<0 >99	unknown error
all others	error No.xx

Table 51: Message returned by strerror()—CLIB runtime environment

The time zone (7.12.1)

The time zone function is not supported.

clock() (7.12.2.1)

The `clock()` function is not supported.

A

- abort
 - implementation-defined behavior 417
 - implementation-defined behavior in C89 (CLIB) 433
 - implementation-defined behavior in C89 (DLIB) 430
 - system termination (DLIB) 137
- absolute location
 - data, placing at (@) 231
 - language support for 196
 - #pragma location 335
- ADC12 module. 74
- addressing. *See* memory types, data models, and code models
- algorithm (STL header file) 374
- alignment 295
 - forcing stricter (#pragma data_alignment) 328
 - in structures (#pragma pack) 339
 - in structures, causing problems 228
 - of an object (__ALIGNOF__) 196
 - of data types. 296
- alignment (pragma directive) 413, 427
- __ALIGNOF__ (operator) 196
- anonymous structures 229
- anonymous symbols, creating 193
- ANSI C. *See* C89
- application
 - building, overview of 54
 - execution, overview of 50
 - startup and termination (CLIB) 163
 - startup and termination (DLIB) 135
- argv (argument), implementation-defined behavior 406
- arrays
 - designated initializers in 193
 - global, accessing 188
 - implementation-defined behavior 410
 - implementation-defined behavior in C89. 425
 - incomplete at end of structs 193
 - non-lvalue 199
 - of incomplete types 198
 - single-value initialization. 199
- asm, __asm (language extension) 170
- assembler code
 - calling from C 175
 - calling from C++ 178
 - inserting inline 169
- assembler directives
 - for call frame information 189
 - using in inline assembler code 169
- assembler instructions, inserting inline 169
- assembler labels
 - default for application startup 54
 - making public (--public_equ). 286
- assembler language interface 167
 - calling convention. *See* assembler code
- assembler list file, generating 274
- assembler output file 177
- asserts
 - implementation-defined behavior of 414
 - implementation-defined behavior of in C89, (CLIB) . . 432
 - implementation-defined behavior of in C89, (DLIB) . . 428
 - including in application 367
- assert.h (CLIB header file) 378
- assert.h (DLIB header file) 372
 - __assignment_by_bitwise_copy_allowed, symbol used in library 377
- @ (operator)
 - placing at absolute address. 231
 - placing in segments 233
- atomic operations 76
 - __monitor 315
- attributes
 - object 311
 - type 309
- auto variables 68
 - at function entrance 182
 - programming hints for efficient code. 240
 - using in inline assembler statements 169

B

backtrace information *See* call frame information

Barr, Michael 33

baseaddr (pragma directive) 413, 427

__BASE_FILE__ (predefined symbol) 362

basic type names, using in preprocessor expressions
(--migration_preprocessor_extensions) 276

basic_template_matching (pragma directive) 325
 using 211

batch files

- error return codes 252
- none for building library from command line 125

__bcd_add_long (intrinsic function) 349

__bcd_add_long_long (intrinsic function) 349

__bcd_add_short (intrinsic function) 349

__bic_SR_register (intrinsic function) 349

__bic_SR_register_on_exit (intrinsic function) 350

binary streams 415

binary streams in C89 (CLIB) 432

binary streams in C89 (DLIB) 429

__bis_GIE_interrupt_state (intrinsic function) 350

bis_nmi_ie1 (pragma directive) 325

__bis_SR_register (intrinsic function) 350

__bis_SR_register_on_exit (intrinsic function) 350

bit negation 241

bitfields

- data representation of 298
- hints 227
- implementation-defined behavior 411
- implementation-defined behavior in C89 425
- non-standard types in 196

bitfields (pragma directive) 326

bits in a byte, implementation-defined behavior 407

bold style, in this guide 34

bool (data type) 296

- adding support for in CLIB 378
- adding support for in DLIB 372, 375

BSLSIGNATURE (segment) 381

building_runtime (pragma directive) 413, 427

__BUILD_NUMBER__ (predefined symbol) 362

C

C and C++ linkage 180

C/C++ calling convention. *See* calling convention

C header files 372

C language, overview 193

call frame information 189

- in assembler list file 177
- in assembler list file (-IA) 274

call frame information, disabling (--no_call_frame_info) . 278

call graph root (stack usage control directive) 397

call stack 189

callee-save registers, stored on stack 68

calling convention

- C++, requiring C linkage 178
- in compiler 179
- overriding default (__cc_version1) 313
- overriding default (__cc_version2) 314

calloc (library function) 69

- See also* heap
- implementation-defined behavior in C89 (CLIB) 433
- implementation-defined behavior in C89 (DLIB) 430

calls (pragma directive) 327

call_graph_root (pragma directive) 327

call-info (in stack usage control file) 402

can_instantiate (pragma directive) 413, 427

cassert (library header file) 374

cast operators

- in Extended EC++ 202, 213
- missing from Embedded C++ 202

casting

- between pointer types 63
- of pointers and integers 302
- pointers to integers, language extension 198

category (in stack usage control file) 401

cctype (DLIB header file) 374

- `__cc_version1` (extended keyword) 313–314
- `cerrno` (DLIB header file) 374
- `cexit` (system termination code)
 - customizing system termination 138
- CFI (assembler directive) 189
- `cfloat` (DLIB header file) 374
- `char` (data type) 296
 - changing default representation (`--char_is_signed`) 262
 - changing representation (`--char_is_unsigned`) 262
 - implementation-defined behavior 408
 - signed and unsigned 297
- character set, implementation-defined behavior 406
- characters
 - implementation-defined behavior 407
 - implementation-defined behavior in C89 422
- character-based I/O 160
- `--char_is_signed` (compiler option) 262
- `--char_is_unsigned` (compiler option) 262
- check that (linker directive) 398
- checksum
 - calculation of 220
- CHECKSUM (segment) 381
- `cinttypes` (DLIB header file) 374
- class memory (extended EC++) 204
- class template partial specialization
 - matching (extended EC++) 210
- CLIB 378
 - documentation 32
 - library reference information for 33
 - naming convention 35
 - runtime environment 159
 - summary of definitions 378
- `--clib` (compiler option) 262
- `climits` (DLIB header file) 375
- `clobber` 170
- `locale` (DLIB header file) 375
- `clock` (CLIB library function),
 - implementation-defined behavior in C89 434
- `clock` (DLIB library function),
 - implementation-defined behavior in C89 431
- `clock` (library function)
 - implementation-defined behavior 418
- `cmath` (DLIB header file) 375
- code
 - execution of 56
 - facilitating for good generation of 239
 - interruption of execution 73
 - verifying linked result 110
- code models 71
 - calling functions in 186
 - configuration 56
 - identifying (`__CODE_MODEL__`) 362
 - selecting (`--code_model`) 263
- code motion (compiler transformation) 238
 - disabling (`--no_code_motion`) 278
- CODE (segment) 382
 - using 107
- codeseq (pragma directive) 413, 427
- `__code_distance` (intrinsic function) 350
- CODE_I (segment) 382
 - linker considerations 108
- CODE_ID (segment) 382
 - linker considerations 108
- `__CODE_MODEL__` (predefined symbol) 362
- `--code_model` (compiler option) 262
- CODE_PAD (segment) 383
- `__code`, symbol used in library 377
- CODE16 (segment) 383
 - using 107
- command line options
 - See also* compiler options
 - part of compiler invocation syntax 249
 - passing 250
 - typographic convention 34
- command prompt icon, in this guide 35
- comments
 - after preprocessor directives 199
 - C++ style, using in C code 193
- common block (call frame information) 189

common subexpr elimination (compiler transformation)	237
disabling (<code>--no_cse</code>)	278
compilation date	
exact time of (<code>__TIME__</code>)	366
identifying (<code>__DATE__</code>)	363
compiler	
environment variables	250
invocation syntax	249
output from	251
compiler listing, generating (<code>-l</code>)	274
compiler object file	47
including debug information in (<code>--debug, -r</code>)	265
output from compiler	251
compiler optimization levels	236
compiler options	255
passing to compiler	250
reading from file (<code>-f</code>)	272
specifying parameters	257
summary	257
syntax	255
for creating skeleton code	177
<code>--warnings_affect_exit_code</code>	252
compiler platform, identifying	364
compiler subversion number	366
compiler transformations	234
compiler version number	366
compiling	
from the command line	54
syntax	249
complex numbers, supported in Embedded C++	202
complex (library header file)	373
complex.h (library header file)	372
compound literals	193
computer style, typographic convention	34
configuration	
basic project settings	54
<code>__low_level_init</code>	138
configuration symbols	
for file input and output	147
for locale	148
for printf and scanf	146
for strtod	150
in library configuration files	125
consistency, module	111
const	
declaring objects	307
non-top level	199
constants, placing in named segment	328
<code>__constrange()</code> , symbol used in library	377
<code>__construction_by_bitwise_copy_allowed</code> , symbol used in library	377
<code>constseg</code> (pragma directive)	328
<code>const_cast</code> (cast operator)	202
contents, of this guide	30
control characters,	
implementation-defined behavior	419
conventions, used in this guide	34
copy initialization, of segments	108
copyright notice	2
<code>__CORE__</code> (predefined symbol)	362
core	
identifying	362
cos (library function)	370
cos (library routine)	133–134
cosf (library routine)	133–134
cosl (library routine)	133–134
<code>__COUNTER__</code> (predefined symbol)	362
<code>__cplusplus</code> (predefined symbol)	362
<code>csetjmp</code> (DLIB header file)	375
<code>csignal</code> (DLIB header file)	375
<code>cspy_support</code> (pragma directive)	413, 427
CSTACK (segment)	383
<i>See also</i> stack	
CSTART (segment)	384
cstartup (system startup code)	
customizing system initialization	138
source files for (CLIB)	163
source files for (DLIB)	135
<code>cstat_disable</code> (pragma directive)	323

- cstat_enable (pragma directive) 323
 - cstat_restore (pragma directive) 323
 - cstat_suppress (pragma directive) 323
 - cstdarg (DLIB header file) 375
 - cstdbool (DLIB header file) 375
 - cstddef (DLIB header file) 375
 - cstdio (DLIB header file) 375
 - cstdlib (DLIB header file) 375
 - cstring (DLIB header file) 375
 - ctime (DLIB header file) 375
 - ctype.h (library header file) 372, 378
 - cwctype.h (library header file) 375
 - ?C_EXIT (assembler label) 165
 - ?C_GETCHAR (assembler label) 165
 - C_INCLUDE (environment variable) 250
 - ?C_PUTCHAR (assembler label) 165
 - C-SPY
 - debug support for C++ 209
 - interface to system termination 138
 - low-level interface (CLIB) 164
 - C-STAT for static analysis, documentation for 32
 - C++
 - See also Embedded C++ and Extended Embedded C++*
 - absolute location 232–233
 - calling convention 178
 - dynamic initialization in 108
 - header files 373
 - language extensions 214
 - standard template library (STL) 374
 - static member variables 232–233
 - support for 40
 - C++ header files 373
 - C++ names, in assembler code 178
 - C++ objects, placing in memory type 65
 - C++ terminology 34
 - C++-style comments 193
 - C89
 - implementation-defined behavior 421
 - support for 193
 - c89 (compiler option) 261
 - C99. *See* Standard C
- ## D
- D (compiler option) 263
 - data
 - alignment of 295
 - different ways of storing 60
 - located, declaring extern 232
 - placing 230, 289, 329
 - at absolute location 231
 - representation of 295
 - storage 59
 - verifying linked result 110
 - data block (call frame information) 189
 - data memory attributes, using 61
 - data models 66
 - configuration 56
 - identifying (`__DATA_MODEL__`) 363
 - Large 68
 - Medium 67
 - setting (`--data_model`) 264
 - Small 67
 - data pointers 301
 - data types 296
 - avoiding large 227
 - floating point 299
 - in C++ 307
 - integer types 296
 - dataseg (pragma directive) 329
 - data_alignment (pragma directive) 328
 - `__DATA_MODEL__` (predefined symbol) 363
 - `--data_model` (compiler option) 264
 - `__data16` (extended keyword) 314
 - DATA16_AC (segment) 384
 - DATA16_AN (segment) 384
 - DATA16_C (segment) 384
 - DATA16_HEAP (segment) 385

DATA16_I (segment)	385	delete operator (extended EC++)	207
DATA16_ID (segment)	385	delete (keyword)	69
DATA16_N (segment)	386	--dependencies (compiler option)	265
DATA16_P (segment)	386	deque (STL header file)	374
__data16_read_addr (intrinsic function)	351	destructors and interrupts, using	208
__data16_size_t	208	device description files, preconfigured for C-SPY	42
DATA16_Z (segment)	386	diagnostic messages	253
__data20 (extended keyword)	314	classifying as compilation errors	266
DATA20_AC (segment)	387	classifying as compilation remarks	266
DATA20_AN (segment)	387	classifying as compiler warnings	267
DATA20_C (segment)	387	disabling compiler warnings	282
DATA20_HEAP (segment)	387	disabling wrapping of in compiler	282
DATA20_I (segment)	388	enabling compiler remarks	288
DATA20_ID (segment)	388	listing all used by compiler	268
__data20_longjmp (intrinsic function)	352	suppressing in compiler	267
DATA20_N (segment)	389	--diagnostics_tables (compiler option)	268
DATA20_P (segment)	389	diagnostics, implementation-defined behavior	405
__data20_read_char (intrinsic function)	352	diag_default (pragma directive)	331
__data20_read_long (intrinsic function)	352	--diag_error (compiler option)	266
__data20_read_short (intrinsic function)	352	diag_error (pragma directive)	331
__data20_setjmp (intrinsic function)	352	--diag_remark (compiler option)	266
__data20_write_char (intrinsic function)	353	diag_remark (pragma directive)	332
__data20_write_long (intrinsic function)	353	--diag_suppress (compiler option)	267
__data20_write_short (intrinsic function)	353	diag_suppress (pragma directive)	332
DATA20_Z (segment)	389	--diag_warning (compiler option)	267
__DATE__ (predefined symbol)	363	diag_warning (pragma directive)	332
date (library function), configuring support for	122	DIFUNCT (segment)	108, 390
--debug (compiler option)	265	directives	
debug information, including in object file	265	pragma	43, 323
decimal point, implementation-defined behavior	419	directory, specifying as parameter	256
declarations		__disable_interrupt (intrinsic function)	353
empty	199	--discard_unused_publics (compiler option)	268
in for loops	193	disclaimer	2
Kernighan & Ritchie	241	DLIB	371
of functions	180	configurations	126
declarations and statements, mixing	193	configuring	124, 269
declarators, implementation-defined behavior in C89	426	documentation	32
define_type_info (pragma directive)	413, 427	naming convention	35
__delay_cycles (intrinsic function)	353	reference information. <i>See</i> the online help system	369

runtime environment 115
 --dlib (compiler option) 268
 --dlib_config (compiler option) 269
 DLib_Defaults.h (library configuration file) 125
 __DLIB_FILE_DESCRIPTOR (configuration symbol) 147
 document conventions 34
 documentation
 contents of this 30
 how to use this 29
 overview of guides 31
 who should read this 29
 domain errors, implementation-defined behavior 414
 domain errors, implementation-defined behavior in C89
 (CLIB) 432
 domain errors, implementation-defined behavior in C89
 (DLIB) 428
 --double (compiler option) 270
 double (data type) 299
 avoiding 227
 configuring size of floating-point type 57
 do_not_instantiate (pragma directive) 413, 427
 dynamic initialization 135, 163
 and C++ 108
 dynamic memory 69

E

-e (compiler option) 270
 early_initialization (pragma directive) 413, 427
 --ec++ (compiler option) 270
 edition, of this guide 2
 --eec++ (compiler option) 271
 Embedded C++ 201
 differences from C++ 202
 enabling 270
 function linkage 180
 language extensions 201
 overview 201
 Embedded C++ Technical Committee 34
 embedded systems, IAR special support for 42
 __embedded_cplusplus (predefined symbol) 363
 __enable_interrupt (intrinsic function) 354
 --enable_multibytes (compiler option) 271
 --enable_restrict (compiler option) 271
 enablig restrict keyword 271
 entry label, program 135
 enumerations
 implementation-defined behavior 411
 implementation-defined behavior in C89 425
 enums
 data representation 297
 forward declarations of 198
 environment
 implementation-defined behavior 406
 implementation-defined behavior in C89 421
 runtime (CLIB) 159
 runtime (DLIB) 115
 environment names, implementation-defined behavior 407
 environment variables
 C_INCLUDE 250
 environment (native),
 implementation-defined behavior 420
 epilogue 336
 EQU (assembler directive) 286
 ERANGE 414
 ERANGE (C89) 428
 errno value at underflow,
 implementation-defined behavior 417
 errno.h (library header file) 372, 378
 error messages 254
 classifying for compiler 266
 error return codes 252
 error (pragma directive) 333
 --error_limit (compiler option) 272
 escape sequences, implementation-defined behavior 407
 __even_in_range (intrinsic function) 354
 example 75
 exception handling, missing from Embedded C++ 202
 exception vectors 107

exclude (stack usage control directive)	398
_Exit (library function)	137
exit (library function)	137
implementation-defined behavior	417
implementation-defined behavior in C89	431, 433
_exit (library function)	137
__exit (library function)	137
exp (library routine)	133
expf (library routine)	133
expl (library routine)	133
export keyword, missing from Extended EC++	209
extended command line file	
for compiler	272
passing options	250
Extended Embedded C++	202
enabling	271
extended keywords	309
enabling (-e)	270
overview	42
summary	312
syntax	
object attributes	312
type attributes on data objects	62, 310
type attributes on functions	311
__ramfunc	79
extern "C" linkage	206

F

-f (compiler option)	272
far (memory type)	61
fatal error messages	254
fdopen, in stdio.h	376
fegettrapdisable	375
fegetrapenable	375
FENV_ACCESS, implementation-defined behavior	410
fenv.h (library header file)	372
additional C functionality	375

fgetpos (library function), implementation-defined behavior	417
fgetpos (library function), implementation-defined behavior in C89	430
field width, library support for	162
__FILE__ (predefined symbol)	363
file buffering, implementation-defined behavior	415
file dependencies, tracking	265
file input and output	
configuration symbols for	147
file paths, specifying for #include files	274
file position, implementation-defined behavior	415
file streams lock interface	155
file systems in C89	432
file (zero-length), implementation-defined behavior	416
filename	
extension for device description files	42
extension for header files	41
extension for linker configuration file	101
of object file	284
search procedure for	250
specifying as parameter	256
filenames (legal), implementation-defined behavior	416
fileno, in stdio.h	376
files, implementation-defined behavior	
handling of temporary	416
multibyte characters in	416
opening	416
float (data type)	299
floating-point constants	
hexadecimal notation	193
hints	228
floating-point environment, accessing or not	343
floating-point expressions	
contracting or not	344
using in preprocessor extensions	276
floating-point format	299
hints	227
implementation-defined behavior	409
implementation-defined behavior in C89	424

- special cases 300
- 32-bits 300
- 64-bits 300
- floating-point numbers, support for in printf formatters . . 161
- floating-point status flags 375
- floating-point type, configuring size of double 57
- float.h (library header file) 372, 378
- FLT_EVAL_METHOD, implementation-defined behavior 409, 414, 418
- FLT_ROUNDS, implementation-defined behavior 409, 418
- fmod (library function), implementation-defined behavior in C89 428, 432
- for loops, declarations in 193
- formats
 - floating-point values 299
 - standard IEEE (floating point) 299
- _formatted_write (library function) 161
- FP_CONTRACT, implementation-defined behavior 410
- fragmentation, of heap memory 70
- FRAM memory 59
- free (library function). *See also* heap 69
- fsetpos (library function), implementation-defined behavior 417
- fstream (library header file) 373
- ftell (library function), implementation-defined behavior
 - in C89 430
- Full DLIB (library configuration) 127
- __func__ (predefined symbol) 200, 364
- __FUNCTION__ (predefined symbol) 200, 364
- function calls
 - calling convention 179
 - eliminating overhead of by inlining 81
 - preserved registers across 181
- function declarations, Kernighan & Ritchie 241
- function entrance, preventing registers from being saved . . 74
- function execution, in RAM 79
- function inlining (compiler transformation) 238
 - disabling (--no_inline) 279
- function pointers 301

- function prototypes 240
 - enforcing 288
- function template parameter deduction (extended EC++) . 210
- function type information, omitting in object output. . . . 283
- function (pragma directive) 413, 427
- function (stack usage control directive) 399
- functional (STL header file) 374
- functions 71
 - calling in different code models 186
 - declaring 180, 240
 - inlining 193, 238, 240, 334
 - interrupt 73, 76
 - intrinsic 167, 240
 - monitor 76
 - omitting type info 283
 - parameters 182
 - placing in memory 230, 233, 289
 - placing segments for 104
 - recursive
 - avoiding 240
 - storing data on stack 69
 - reentrancy (DLIB) 370
 - related extensions 71
 - return values from 183
 - special function types 73
 - verifying linked result 110
- function_effects (pragma directive) 413, 427
- function-spec (in stack usage control file) 401

G

- getchar (library function) 161
- getw, in stdio.h 376
- getzone (library function), configuring support for 122
- __get_interrupt_state (intrinsic function) 354
- __get_R4_register (intrinsic function) 355
- __get_R5_register (intrinsic function) 355
- __get_SP_register (intrinsic function) 355
- __get_SR_register (intrinsic function) 356

<code>__get_SR_register_on_exit</code> (intrinsic function)	356
global arrays, accessing	188
global variables	
accessing	188
handled during system termination	137
hints for not using	240
initialized during system startup	136
<code>--guard_calls</code> (compiler option)	272
guidelines, reading	29

H

Harbison, Samuel P.	33
hardware multiplier	150
enabling on command line	277
hardware support in compiler	116
<code>hash_map</code> (STL header file)	374
<code>hash_set</code> (STL header file)	374
<code>__has_constructor</code> , symbol used in library	377
<code>__has_destructor</code> , symbol used in library	377
<code>hdrstop</code> (pragma directive)	413, 427
header files	
C	372
C++	373
library	370
special function registers	243
STL	374
<code>DLib_Defaults.h</code>	125
including <code>stdbool.h</code> for <code>bool</code>	297
including <code>stddef.h</code> for <code>wchar_t</code>	297
header names, implementation-defined behavior	412
<code>--header_context</code> (compiler option)	273
heap	
dynamic memory	69
segments for	106
storing data	60
VLA allocated on	292
heap segments	
CLIB	218

DATA16_HEAP (segment)	385
DATA20_HEAP (segment)	387
DLIB	218
placing	106
heap size	
and standard I/O	218
changing default	105–106
HEAP (segment)	218
heap (zero-sized), implementation-defined behavior	417
hints	
for good code generation	239
implementation-defined behavior	411
using efficient data types	227
<code>--hw_workaround</code> (compiler option)	273

I

<code>-I</code> (compiler option)	274
IAR Command Line Build Utility	125
IAR Systems Technical Support	254
<code>iarbuild.exe</code> (utility)	125
<code>__iar_cos_accurate</code> (library routine)	134
<code>__iar_cos_accuratef</code> (library routine)	134
<code>__iar_cos_accuratef</code> (library function)	370
<code>__iar_cos_accuratel</code> (library routine)	134
<code>__iar_cos_accuratel</code> (library function)	370
<code>__iar_cos_small</code> (library routine)	133
<code>__iar_cos_smallf</code> (library routine)	133
<code>__iar_cos_smallll</code> (library routine)	133
<code>__IAR_DLIB_PERTHREAD_INIT_SIZE</code> (macro)	156
<code>__IAR_DLIB_PERTHREAD_SIZE</code> (macro)	156
<code>__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET</code>	
(symbolptr)	156
<code>__iar_exp_small</code> (library routine)	133
<code>__iar_exp_smallf</code> (library routine)	133
<code>__iar_exp_smallll</code> (library routine)	133
<code>__iar_log_small</code> (library routine)	133
<code>__iar_log_smallf</code> (library routine)	133
<code>__iar_log_smallll</code> (library routine)	133

- `__iar_log10_small` (library routine) 133
- `__iar_log10_smallf` (library routine) 133
- `__iar_log10_smallll` (library routine) 133
- `__iar_Powf` (library routine) 134
- `__iar_Powl` (library routine) 134
- `__iar_Pow_accurate` (library routine) 134
- `__iar_pow_accurate` (library routine) 134
- `__iar_Pow_accuratef` (library routine) 134
- `__iar_pow_accuratef` (library routine) 134
- `__iar_pow_accuratef` (library function) 370
- `__iar_Pow_accuratel` (library routine) 134
- `__iar_pow_accuratel` (library routine) 134
- `__iar_pow_accuratel` (library function) 370
- `__iar_pow_small` (library routine) 133
- `__iar_pow_smallf` (library routine) 133
- `__iar_pow_smallll` (library routine) 133
- `__iar_program_start` (label) 135
- `__iar_Sin` (library routine) 133
- `__iar_Sinf` (library routine) 134
- `__iar_Sinl` (library routine) 134
- `__iar_Sin_accurate` (library routine) 134
- `__iar_sin_accurate` (library routine) 134
- `__iar_Sin_accuratef` (library routine) 134
- `__iar_sin_accuratef` (library routine) 134
- `__iar_sin_accuratef` (library function) 370
- `__iar_Sin_accuratel` (library routine) 134
- `__iar_sin_accuratel` (library routine) 134
- `__iar_sin_accuratel` (library function) 370
- `__iar_Sin_small` (library routine) 133
- `__iar_sin_small` (library routine) 133
- `__iar_Sin_smallf` (library routine) 133
- `__iar_sin_smallf` (library routine) 133
- `__iar_Sin_smallll` (library routine) 133
- `__iar_sin_smallll` (library routine) 133
- `__IAR_SYSTEMS_ICC__` (predefined symbol) 364
- `__iar_tan_accurate` (library routine) 134
- `__iar_tan_accuratef` (library routine) 134
- `__iar_tan_accuratef` (library function) 370
- `__iar_tan_accuratel` (library routine) 134
- `__iar_tan_accuratel` (library function) 370
- `__iar_tan_small` (library routine) 133
- `__iar_tan_smallf` (library routine) 133
- `__iar_tan_smallll` (library routine) 133
- `iccbutl.h` (library header file) 378
- icons, in this guide 35
- IDE
 - building a library from 124
 - overview of build tools 39
- identifiers, implementation-defined behavior 407
- identifiers, implementation-defined behavior in C89 422
- IEEE format, floating-point values 299
- `important_typedef` (pragma directive) 413, 427
- include files
 - including before source files 285
 - specifying 250
- `include_alias` (pragma directive) 333
- infinity 300
- infinity (style for printing), implementation-defined behavior 416
- INFO (segment) 390
- INFOA (segment) 390
- INFOB (segment) 390
- INFOC (segment) 391
- INFOD (segment) 391
- information memory, linker segment for 390–391
- inheritance, in Embedded C++ 201
- initialization
 - dynamic 135, 163
 - single-value 199
- initializers, static 198
- inline assembler 169
 - avoiding 240
 - See also* assembler language interface
- inline functions 193
 - in compiler 238
- inline (pragma directive) 334
- inlining functions 81
 - implementation-defined behavior 411
- installation directory 34

instantiate (pragma directive)	413, 427
int (data type) signed and unsigned	296
integer types	296
casting	302
implementation-defined behavior	409
intptr_t	302
ptrdiff_t	302
size_t	302
uintptr_t	303
integers, implementation-defined behavior in C89	423
integral promotion	241
Intellectual Property Encapsulation (IPE)	151
internal error	254
__interrupt (extended keyword)	74, 315
using in pragma directives	346
interrupt functions	73
for MSP430X.	75
placement in memory.	107
interrupt handler. <i>See</i> interrupt service routine	
interrupt service routine	73
interrupt state, restoring	358
interrupt vector	73
specifying with pragma directive	345
Interrupt Vector Generators, writing interrupt service routines for	74
interrupt vector table	73
in linker configuration file	107
INTVEC segment	391
interrupts	
disabling	315
during function execution	76
processor state	68
using with EC++ destructors	208
intptr_t (integer type)	302
__intrinsic (extended keyword).	315
intrinsic functions	240
overview	167
summary	347
intrinsics.h (header file)	347
inttypes.h (library header file).	372

INTVEC (segment).	107, 391
intwri.c (library source code)	162
invocation syntax	249
iomanip (library header file)	373
ios (library header file)	373
iosfwd (library header file)	373
iostream (library header file).	373
IPE (Intellectual Property Encapsulation).	151
IPECODE16 (segment).	392
IPEDATA16_C (segment)	392
IPEDATA16_N (segment)	393
IPESIGNATURE (segment).	393
IPE_B1 (segment).	391
IPE_B2 (segment).	392
iso646.h (library header file).	372
ISR_CODE (segment)	393
using	107
istream (library header file).	373
italic style, in this guide	34–35
iterator (STL header file)	374
I/O register. <i>See</i> SFR	
I/O, character-based	160
I2C (I2CIV) module	74

J

Josuttis, Nicolai M.	33
JTAGSIGNATURE (segment)	393

K

keep_definition (pragma directive)	413, 427
Kernighan & Ritchie function declarations	241
disallowing	288
Kernighan, Brian W.	33
keywords	309
extended, overview of	42

- L**
- l (compiler option) 274
 - for creating skeleton code 177
 - labels 199
 - assembler, making public 286
 - __iar_program_start 135
 - __program_start 135
 - Labrosse, Jean J. 33
 - Lajoie, Josée 33
 - language extensions
 - Embedded C++ 201
 - enabling using pragma 334
 - enabling (-e) 270
 - language overview 40
 - language (pragma directive) 334
 - Large (code model) 72
 - Large (data model) 68
 - _large_write (library function) 161
 - libraries
 - reason for using 48
 - standard template library 374
 - using a prebuilt 127
 - using a prebuilt (CLIB) 159
 - library configuration files
 - DLIB 126
 - DLib_Defaults.h 125
 - modifying 125
 - specifying 269
 - library documentation 369
 - library features, missing from Embedded C++ 202
 - library functions
 - summary, CLIB 378
 - summary, DLIB 372
 - online help for 33
 - library header files 370
 - library modules
 - creating 275
 - overriding 123
 - library object files 370
 - library project, building using a template 124
 - library_default_requirements (pragma directive) 413, 427
 - library_module (compiler option) 275
 - library_provides (pragma directive) 413, 427
 - library_requirement_override (pragma directive) 413, 428
 - lightbulb icon, in this guide 35
 - limits.h (library header file) 372, 378
 - __LINE__ (predefined symbol) 364
 - linkage, C and C++ 180
 - linker 85
 - linker configuration file 88
 - for placing code and data 88
 - in depth 397
 - overview of 397
 - using the -P command 103
 - using the -Z command 103
 - linker map file 110
 - linker options
 - typographic convention 34
 - Q 108
 - linker segment. *See* segment
 - linking
 - from the command line 54
 - in the build process 48
 - introduction 85
 - process for 87
 - Lippman, Stanley B. 33
 - list (STL header file) 374
 - listing, generating 274
 - literals, compound 193
 - literature, recommended 33
 - local variables, *See* auto variables
 - locale
 - adding support for in library 149
 - changing at runtime 149
 - implementation-defined behavior 408, 419
 - removing support for 149
 - support for 147

locale.h (library header file)	372
located data segments	104
located data, declaring extern	232
location (pragma directive)	231, 335
--lock_r4 (compiler option)	275
--lock_r5 (compiler option)	275
log (library routine)	133
logf (library routine)	133
logl (library routine)	133
log10 (library routine)	133
log10f (library routine)	133
log10l (library routine)	133
long double (data type)	299
long float (data type), synonym for double	198
long long (data type)	
avoiding	227
restrictions	297
long long (data type) signed and unsigned	296
long (data type) signed and unsigned	296
longjmp, restrictions for using	371
loop unrolling (compiler transformation)	238
disabling	282
#pragma unroll	345
loop-invariant expressions	238
__low_level_init	136
customizing	138
initialization phase	50
low_level_init.c	135, 163
__low_power_mode_n (intrinsic function)	356
__low_power_mode_off_on_exit (intrinsic function)	356
low-level processor operations	194
accessing	167

M

macros	
embedded in #pragma optimize	338
ERANGE (in errno.h)	414, 428
inclusion of assert	367

NULL, implementation-defined behavior	415
in C89 for CLIB	431
in C89 for DLIB	428
substituted in #pragma directives	194
variadic	193
--macro_positions_in_diagnostics (compiler option)	276
main (function)	
definition (C89)	421
implementation-defined behavior	406
malloc (library function)	
<i>See also</i> heap	69
implementation-defined behavior in C89	430, 433
Mann, Bernhard	33
map (STL header file)	374
map, linker	110
math functions rounding mode, implementation-defined behavior	418
math functions (library functions)	131
MathLib	134
math.h (library header file)	372, 378
max recursion depth (stack usage control directive)	399
MB_LEN_MAX, implementation-defined behavior	418
Medium (data model)	67
__medium_write (library function)	161
memory	
accessing	56, 60, 187
using data16 method	188
using data20 method	188
allocating in C++	69
dynamic	69
heap	69
non-initialized	245
RAM, saving	240
releasing in C++	69
stack	68
saving	240
used by global or static variables	60
memory clobber	170
memory consumption, reducing	161
memory management, type-safe	201

- memory map
 - initializing SFRs 138
 - linker configuration for 101
 - memory placement
 - of linker segments 88
 - using type definitions 63
 - Memory Protection Unit (MPU) 150
 - memory segment. *See* segment
 - memory types 60
 - C++ 65
 - placing variables in 65
 - pointers 63
 - specifying 61
 - structures 64
 - summary 61
 - memory (pragma directive) 413, 428
 - memory (STL header file) 374
 - `__memory_of`
 - operator 205
 - symbol used in library 377
 - message (pragma directive) 336
 - messages
 - disabling 290
 - forcing 336
 - Meyers, Scott 33
 - `--mfc` (compiler option) 276
 - `--migration_preprocessor_extensions` (compiler option) 276
 - migration, from earlier IAR compilers 32
 - MISRA C
 - documentation 32
 - `--misrac` (compiler option) 259
 - `--misrac_verbose` (compiler option) 259
 - `--misrac1998` (compiler option) 259
 - `--misrac2004` (compiler option) 259
 - mode changing, implementation-defined behavior 416
 - module consistency 111
 - `rtmodel` 341
 - module map, in linker map file 111
 - module name, specifying (`--module_name`) 277
 - module summary, in linker map file 111
 - `--module_name` (compiler option) 277
 - `module_name` (pragma directive) 413, 428
 - module-spec (in stack usage control file) 402
 - `__monitor` (extended keyword) 315
 - monitor functions 76, 315
 - MPU (Memory Protection Unit) 150
 - MPU_B1 (segment) 394
 - MPU_B2 (segment) 394
 - MSP430
 - information memory 390–391
 - memory access 56
 - MSP430X
 - code models 71
 - data models 66
 - interrupt functions 75
 - support for 41
 - multibyte character support 271
 - multibyte characters, implementation-defined
 - behavior 407, 419
 - multiple inheritance
 - in Extended EC++ 202
 - missing from Embedded C++ 202
 - `--multiplier` (compiler option) 277
 - multithreaded environment 152
 - multi-file compilation 235
 - mutable attribute, in Extended EC++ 202, 214
- ## N
- name (in stack usage control file) 402
 - names block (call frame information) 189
 - namespace support
 - in Extended EC++ 202, 214
 - missing from Embedded C++ 202
 - naming conventions 35
 - NaN
 - implementation of 300
 - implementation-defined behavior 416

native environment,	
implementation-defined behavior	420
NDEBUG (preprocessor symbol)	367
near (memory type)	61
new operator (extended EC++)	207
new (keyword)	69
new (library header file)	373
no calls from (stack usage control directive)	400
non-initialized variables, hints for	245
non-scalar parameters, avoiding	240
NOP (assembler instruction)	356
__noreturn (extended keyword)	318
Normal DLIB (library configuration)	127
Not a number (NaN)	300
__no_alloc (extended keyword)	316
__no_alloc_str (operator)	316
__no_alloc_str16 (operator)	316
__no_alloc16 (extended keyword)	316
--no_call_frame_info (compiler option)	278
--no_code_motion (compiler option)	278
no_crosscall (#pragma optimize parameter)	338
no_crossjump (#pragma optimize parameter)	338
--no_cse (compiler option)	278
no_epilogue (pragma directive)	336
__no_init (extended keyword)	245, 317
--no_inline (compiler option)	279
__no_multiplier (extended keyword)	317
__no_operation (intrinsic function)	356
--no_path_in_file_macros (compiler option)	279
no_pch (pragma directive)	413, 428
__no_pic (extended keyword)	318
--no_rw_dynamic_init (compiler option)	279
--no_size_constraints (compiler option)	280
--no_static_destruction (compiler option)	280
--no_system_include (compiler option)	280
--no_tbaa (compiler option)	281
--no_typedefs_in_diagnostics (compiler option)	281
--no_ubrof_messages (compiler option)	282
--no_unroll (compiler option)	282
--no_warnings (compiler option)	282

--no_wrap_diagnostics (compiler option)	282
NULL	
implementation-defined behavior	415
implementation-defined behavior in C89 (CLIB)	431
implementation-defined behavior in C89 (DLIB)	428
in library header file (CLIB)	378
pointer constant, relaxation to Standard C	198
numeric conversion functions,	
implementation-defined behavior	420
numeric (STL header file)	374

O

-O (compiler option)	283
-o (compiler option)	284
object attributes	311
object filename, specifying (-o)	284
object module name, specifying (--module_name)	277
object_attribute (pragma directive)	245, 337
offsetof	378
--omit_types (compiler option)	283
once (pragma directive)	413, 428
--only_stdout (compiler option)	284
operators	
<i>See also</i> @ (operator)	
for cast	
in Extended EC++	202
missing from Embedded C++	202
for segment control	197
new and delete	207
precision for 32-bit float	300
precision for 64-bit float	300
sizeof, implementation-defined behavior	419
variants for cast	213
_Pragma (preprocessor)	193
__ALIGNOF__, for alignment control	196
__memory_of.	205
?, language extensions for	215

- optimization
 - code motion, disabling 278
 - common sub-expression elimination, disabling 278
 - configuration 57
 - disabling 237
 - function inlining, disabling (`--no_inline`) 279
 - hints 239
 - loop unrolling, disabling 282
 - specifying (`-O`) 283
 - techniques 237
 - type-based alias analysis, disabling (`--tbaa`) 281
 - using pragma directive 337
 - optimization levels 236
 - optimize (pragma directive) 337
 - option parameters 255
 - options, compiler. *See* compiler options
 - `__op_code` (intrinsic function) 357
 - Oram, Andy 33
 - ostream (library header file) 373
 - output
 - from preprocessor 285
 - specifying for linker 54
 - supporting non-standard 162
 - `--output` (compiler option) 284
 - overhead, reducing 238
- ## P
- pack (pragma directive) 303, 339
 - packed structure types 303
 - parameters
 - function 182
 - hidden 182
 - non-scalar, avoiding 240
 - register 182
 - rules for specifying a file or directory 256
 - specifying 257
 - stack 182–183
 - typographic convention 34
 - part number, of this guide 2
 - `--pending_instantiations` (compiler option) 284
 - permanent registers 181
 - perorr (library function),
 - implementation-defined behavior in C89 430, 433
 - `__persistent` (extended keyword) 318
 - cannot be used with ROPI 80
 - placement
 - in named segments 233
 - of code and data, introduction to 88
 - plain char, implementation-defined behavior 408
 - pointer types 301
 - differences between 63
 - mixing 198
 - pointers
 - casting 63, 302
 - data 301
 - function 301
 - implementation-defined behavior 410
 - implementation-defined behavior in C89 425
 - polymorphism, in Embedded C++ 201
 - porting, code containing pragma directives 325
 - `__POSITION_INDEPENDENT_CODE__` (predefined symbol) 364
 - position-independent code and read-only data (ROPI) 80
 - possible calls (stack usage control directive) 400
 - pow (library routine) 133–134
 - alternative implementation of 370
 - powf (library routine) 133–134
 - powl (library routine) 133–134
 - pragma directives 43
 - summary 323
 - `basic_template_matching`, using 211
 - for absolute located data 231
 - list of all recognized 413
 - list of all recognized (C89) 427
 - pack 303, 339
 - `_Pragma` (preprocessor operator) 193
 - precision arguments, library support for 162

predefined symbols	
overview	43
summary	362
--predef_macro (compiler option)	285
--preinclude (compiler option)	285
--preprocess (compiler option)	285
preprocessor	
operator (_Pragma)	193
output	285
preprocessor directives	
comments at the end of	199
implementation-defined behavior	412
implementation-defined behavior in C89	426
#pragma	323
preprocessor extensions	
compatibility	276
__VA_ARGS__	193
#warning message	367
preprocessor symbols	362
defining	263
preserved registers	181
__PRETTY_FUNCTION__ (predefined symbol)	365
primitives, for special functions	73
print formatter, selecting	130
printf (library function)	129, 161
choosing formatter	129
configuration symbols	146
customizing	162
implementation-defined behavior	417
implementation-defined behavior in C89	430, 433
selecting	162
__printf_args (pragma directive)	340
printing characters, implementation-defined behavior	419
processor configuration	55
processor operations	
accessing	167
low-level	194
program entry label	135
program termination, implementation-defined behavior	406

programming hints	239
__program_start (label)	135
projects	
basic settings for	54
setting up for a library	124
prototypes, enforcing	288
ptrdiff_t (integer type)	302, 378
PUBLIC (assembler directive)	286
publication date, of this guide	2
--public_equ (compiler option)	286
public_equ (pragma directive)	339
putchar (library function)	161
putenv (library function), absent from DLIB	142
putw, in stdio.h	376

Q

-Q (linker option)	108
qualifiers	
const and volatile	305
implementation-defined behavior	412
implementation-defined behavior in C89	426
queue (STL header file)	374

R

-r (compiler option)	265
RAM	
function execution in	79
initializers copied from ROM	52
saving memory	240
__ramfunc (extended keyword)	319
cannot be used with ROPI	80
range errors, in linker	110
__raw (extended keyword)	319
example	74
read formatter, selecting	131, 163
reading guidelines	29
reading, recommended	33

- realloc (library function) 69
 - implementation-defined behavior in C89 430, 433
 - See also* heap
- recursive functions
 - avoiding 240
 - storing data on stack 69
- reduce_stack_usage (compiler option) 286
- reentrancy (DLIB) 370
- reference information, typographic convention 35
- register keyword, implementation-defined behavior 411
- register parameters 182
- registered trademarks 2
- registers
 - assigning to parameters 183
 - callee-save, stored on stack 68
 - for function returns 184
 - implementation-defined behavior in C89 425
 - in assembler-level routines 179
 - not saving at function entrance 74
 - preserved 181
- R4
 - excluding from use (--lock_R4) 275
 - getting the value of (__get_R4_register) 355
 - reserving for register variables (--regvar_R4) 286
 - writing to (__set_R4_register) 358
- R5
 - excluding from use (--lock_R5) 275
 - getting the value of (__get_R5_register) 355
 - reserving for register variables (--regvar_R5) 287
 - writing to (__set_R5_register) 358
- scratch 181
- SP
 - getting the value of (__get_SP_register) 355
 - writing to (__set_SP_register) 359
- SR
 - getting the value of on exit 356
 - getting the value of (__get_SR_register) 356
- __REGISTER_MODEL__ (predefined symbol) 365
- __REGISTER_R4__ (predefined symbol) 365
- __REGISTER_R5__ (predefined symbol) 365
- REGVAR_AN (segment) 394
- regvar_r4 (compiler option) 286
- regvar_r5 (compiler option) 287
- reinterpret_cast (cast operator) 202
- relaxed_fp (compiler option) 287
- remark (diagnostic message) 253
 - classifying for compiler 266
 - enabling in compiler 288
- remarks (compiler option) 288
- remove (library function)
 - implementation-defined behavior 416
 - implementation-defined behavior in C89 (CLIB) 433
 - implementation-defined behavior in C89 (DLIB) 430
- remquo, magnitude of 415
- rename (library function)
 - implementation-defined behavior 416
 - implementation-defined behavior in C89 (CLIB) 433
 - implementation-defined behavior in C89 (DLIB) 430
- __ReportAssert (library function) 144
- required (pragma directive) 340
- require_prototypes (compiler option) 288
- RESET (segment) 394
- restrict keyword, enabling 271
- return values, from functions 183
- Ritchie, Dennis M. 33
- __root (extended keyword) 320
- ROPI 80
- __ROPI__ (predefined symbol) 365
- ropi (compiler option) 288
- routines, time-critical 167, 194
- __ro_placement (extended keyword) 320
- rtmodel (assembler directive) 112
- rtmodel (pragma directive) 341
- rtti support, missing from STL 203
- __rt_version (runtime model attribute) 113
- runtime environment
 - CLIB 159
 - DLIB 115

setting up (DLIB)	122
runtime libraries (CLIB)	
introduction	369
filename syntax	160
using prebuilt	159
runtime libraries (DLIB)	
introduction	369
customizing system startup code	138
filename syntax	128
overriding modules in	123
using prebuilt	127
runtime model attributes	111
runtime model definitions	341
runtime type information, missing from Embedded C++	202
R4. <i>See</i> registers	
R5. <i>See</i> registers	

S

__saturated_add_signed_type (intrinsic function)	357
__saturated_add_unsigned_type (intrinsic function)	357
__saturated_sub_signed_type (intrinsic function)	357
__saturated_sub_unsigned_type (intrinsic function)	358
--save_reg20 (compiler option)	289
scanf (library function)	
choosing formatter (CLIB)	162
choosing formatter (DLIB)	130
configuration symbols	146
implementation-defined behavior	417
implementation-defined behavior in C89	433
implementation-defined behavior in C89 (CLIB)	433
implementation-defined behavior in C89 (DLIB)	430
__scanf_args (pragma directive)	341
scratch registers	181
section (pragma directive)	342
segment group name	90
segment map, in linker map file	110
segment memory types, in XLINK	86
--segment (compiler option)	289

segment (pragma directive)	342
segments	
allocation of	88
CODE	107
CODE16	107
declaring (#pragma segment)	342
definition of	86
HEAP	218
ISR_CODE, for interrupt functions (MSP430X)	107
located data	104
naming	90
packing in memory	103
placing in sequence	103
specifying (--segment)	289
summary	379
too long for address range	110
too long, in linker	110
__segment_begin (extended operator)	197
__segment_end (extended operator)	197
__segment_size (extended operator)	197
semaphores	
C example	76
C++ example	78
operations on	315
set (STL header file)	374
setjmp.h (library header file)	372, 378
setlocale (library function)	149
settings, basic for project configuration	54
__set_interrupt_state (intrinsic function)	358
__set_R4_register (intrinsic function)	358
__set_R5_register (intrinsic function)	358
__set_SP_register (intrinsic function)	359
severity level, of diagnostic messages	253
specifying	254
SFR	
accessing special function registers	243
declaring extern special function registers	232
shared object	252
short (data type)	296

- signal (library function)
 - implementation-defined behavior 415
 - implementation-defined behavior in C89 429
- signals, implementation-defined behavior 406
 - at system startup 407
- signal.h (library header file) 372
- SIGNATURE (segment) 395
- signed char (data type) 296–297
 - specifying 262
- signed int (data type) 296
- signed long long (data type) 296
- signed long (data type) 296
- signed short (data type) 296
- silent (compiler option) 290
- silent operation, specifying in compiler 290
- sin (library function) 370
- sin (library routine) 133–134
- sinf (library routine) 133–134
- sinl (library routine) 133–134
- 64-bits (floating-point format) 300
- size (in stack usage control file) 403
- sizeof, using in preprocessor extensions 276
- size_t (integer type) 302, 378
- skeleton code, creating for assembler language interface . 176
- slist (STL header file) 374
- Small (code model) 72
- Small (data model) 67
- _small_write (library function) 162
- source files, list all referred 273
- space characters, implementation-defined behavior 415
- special function registers (SFR) 243
- special function types 73
- sprintf (library function) 129, 161
 - choosing formatter 129
 - customizing 162
- SP. *See* registers
- sscanf (library function)
 - choosing formatter (CLIB) 162
 - choosing formatter (DLIB) 130
- sstream (library header file) 373
- stack 68
 - advantages and problems using 69
 - cleaning after function return 184
 - contents of 68
 - layout 183
 - saving space 240
 - setting up 105
 - size 217
- stack parameters 182–183
- stack pointer 68
- stack segments
 - CSTACK 383
 - placing 105
- stack (STL header file) 374
- stack-size (in stack usage control file) 403
- Standard C 271
 - library compliance with 369
 - specifying strict usage 290
- standard error
 - redirecting in compiler 284
 - See also* diagnostic messages 252
- standard output
 - specifying in compiler 284
- standard template library (STL)
 - in C++ 374
 - in Extended EC++ 202, 210
 - missing from Embedded C++ 202
- startup code 107
 - cstartup 138
 - placement of 107
- startup system. *See* system startup
- statements, implementation-defined behavior in C89 426
- static analysis
 - documentation for 32
- static data, in configuration file 104
- static variables 60
 - taking the address of 240
- static_assert() 196

static_cast (cast operator)	202	strings, supported in Embedded C++	202
status flags for floating-point	375	string.h (library header file)	372, 378
std namespace, missing from EC++		string.h, additional C functionality	376
and Extended EC++	214	strncasecmp, in string.h.	376
stdarg.h (library header file)	372, 378	strlen, in string.h	376
stdbool.h (library header file)	297, 372, 378	Stroustrup, Bjarne	33
__STDC__ (predefined symbol)	365	strstream (library header file)	373
STDC CX_LIMITED_RANGE (pragma directive)	343	strtod (library function), configuring support for	150
STDC FENV_ACCESS (pragma directive)	343	structure types	
STDC FP_CONTRACT (pragma directive)	344	alignment	303
__STDC_VERSION__ (predefined symbol)	366	layout of	303
stddef.h (library header file)	297, 372, 378	packed	303
stderr	122, 284	structures	
stdin	122	accessing using a pointer	188
implementation-defined behavior in C89 (CLIB)	432	aligning	339
implementation-defined behavior in C89 (DLIB)	429	anonymous	196, 229
stdint.h (library header file)	372, 375	implementation-defined behavior	411
stdio.h (library header file)	372, 378	implementation-defined behavior in C89	425
stdio.h, additional C functionality	376	packing and unpacking	229
stdlib.h (library header file)	372, 378	placing in memory type	64
stdout	122, 284	subnormal numbers	299
implementation-defined behavior	415	support, technical	254
implementation-defined behavior in C89 (CLIB)	432	Sutter, Herb	33
implementation-defined behavior in C89 (DLIB)	429	__swap_bytes (intrinsic function)	359
Steele, Guy L.	33	switch statements, hints for using	245
STL	210	SWPB (assembler instruction)	359
strcasecmp, in string.h	376	symbol names, using in preprocessor extensions	277
strdup, in string.h	376	symbols	
streambuf (library header file)	373	anonymous, creating	193
streams		including in output	340
implementation-defined behavior	406	listing in linker map file	111
supported in Embedded C++	202	overview of predefined	43
strerror (library function)		preprocessor, defining	263
implementation-defined behavior in C89 (CLIB)	434	syntax	
strerror (library function), implementation-defined		command line options	255
behavior	420	extended keywords	62, 310–312
strerror (library function),		invoking compiler	249
implementation-defined behavior in C89 (DLIB)	431	system function, implementation-defined behavior.	407, 417
--strict (compiler option)	290	system locks interface	155
string (library header file)	373		

- system startup
 - CLIB 163
 - customizing 138
 - DLIB 135
 - initialization phase 50
 - system termination
 - CLIB 164
 - C-SPY interface to 138
 - DLIB 137
 - system (library function)
 - implementation-defined behavior in C89 434
 - implementation-defined behavior in C89 (DLIB) 431
 - system_include (pragma directive) 413, 428
 - system_include_dir (compiler option) 291
- T**
- tan (library function) 370
 - tan (library routine) 133–134
 - tanf (library routine) 133–134
 - tanl (library routine) 133–134
 - __task (extended keyword) 321
 - technical support, IAR Systems 254
 - template support
 - in Extended EC++ 202, 209
 - missing from Embedded C++ 202
 - Terminal I/O window
 - making available (CLIB) 165
 - not supported when 123–124
 - termination of system. *See* system termination
 - termination status, implementation-defined behavior 417
 - terminology 34
 - Texas Instruments MathLib 134
 - tgmath.h (library header file) 372
 - 32-bits (floating-point format) 300
 - this (pointer) 178
 - class memory 204
 - referring to a class object 204
 - threaded environment 152
 - thread-local storage 155
 - __TIME__ (predefined symbol) 366
 - time zone (library function)
 - implementation-defined behavior in C89 431, 434
 - time zone (library function), implementation-defined behavior 418
 - Timer A (TAIV) module 74
 - Timer B (TBIV) module 74
 - __TIMESTAMP__ (predefined symbol) 366
 - time-critical routines 167, 194
 - time.h (library header file) 372
 - additional C functionality 376
 - time32 (library function), configuring support for 122
 - time64 (library function), configuring support for 122
 - tips, programming 239
 - TLS handling 155
 - TLS16_I (segment) 395
 - TLS16_ID (segment) 395
 - tools icon, in this guide 35
 - trademarks 2
 - transformations, compiler 234
 - translation
 - implementation-defined behavior 405
 - implementation-defined behavior in C89 421
 - trap vectors, specifying with pragma directive 345
 - type attributes 309
 - specifying 344
 - type definitions, used for specifying memory storage 63
 - type information, omitting 283
 - type qualifiers
 - const and volatile 305
 - implementation-defined behavior 412
 - implementation-defined behavior in C89 426
 - typedefs
 - excluding from diagnostics 281
 - repeated 198
 - using in preprocessor extensions 277
 - type_attribute (pragma directive) 344
 - type-based alias analysis (compiler transformation) 238
 - disabling 281

type-safe memory management	201
typographic conventions	34

U

UBROF

format of linkable object files	251
specifying, example of	54
uchar.h (library header file)	372
uintptr_t (integer type)	303
underflow errors, implementation-defined behavior	414–415
underflow range errors, implementation-defined behavior in C89	428, 432
__ungetchar, in stdio.h	376
unions	
anonymous	196, 229
implementation-defined behavior	411
implementation-defined behavior in C89	425
universal character names, implementation-defined behavior	412
unroll (pragma directive)	345
unsigned char (data type)	296–297
changing to signed char	262
unsigned int (data type)	296
unsigned long long (data type)	296
unsigned long (data type)	296
unsigned short (data type)	296
--use_c++_inline (compiler option)	291
utility (STL header file)	374

V

variable type information, omitting in object output	283
variables	
auto	68
defined inside a function	68
global	
accessing	188
placement in memory	60

hints for choosing	240
local. <i>See</i> auto variables	
non-initialized	245
omitting type info	283
placing at absolute addresses	233
placing in named segments	233
static	
placement in memory	60
taking the address of	240
variadic macros	197
vector (pragma directive)	74, 345
cannot be used with ROPI	80
vector (STL header file)	374
version	
compiler subversion number	366
identifying C standard in use (__STDC_VERSION__)	366
of compiler (__VER__)	366
of this guide	2
version1 calling convention	179
version2 calling convention	179
--version (compiler option)	291
--vla (compiler option)	292
void, pointers to	198
volatile	
and const, declaring objects	306
declaring objects	305
protecting simultaneously accesses variables	242
rules for access	306

W

#warning message (preprocessor extension)	367
warnings	253
classifying in compiler	267
disabling in compiler	282
exit code in compiler	292
warnings icon, in this guide	35
warnings (pragma directive)	413, 428
--warnings_affect_exit_code (compiler option)	252, 292

--warnings_are_errors (compiler option) 292
 --warn_about_c_style_casts (compiler option) 292
 wchar_t (data type), adding support for in C 297
 wchar.h (library header file) 372, 375
 wctype.h (library header file) 373
 weak (pragma directive) 346
 web sites, recommended 33
 white-space characters, implementation-defined behavior 406
 write formatter, selecting 162–163
 __write_array, in stdio.h 376
 __write_buffered (DLIB library function) 121

X

XLINK errors

 range error 110
 segment too long 110
 XLINK segment memory types 86
 XLINK. *See* linker

Symbols

. 364
 _Exit (library function) 137
 _exit (library function) 137
 _formatted_write (library function) 161
 _large_write (library function) 161
 _medium_write (library function) 161
 _small_write (library function) 162
 __ALIGNOF__ (operator) 196
 __asm (language extension) 170
 __assignment_by_bitwise_copy_allowed, symbol used
 in library 377
 __BASE_FILE__ (predefined symbol) 362
 __bcd_add_long (intrinsic function) 349
 __bcd_add_long_long (intrinsic function) 349
 __bcd_add_short (intrinsic function) 349
 __bic_SR_register (intrinsic function) 349
 __bic_SR_register_on_exit (intrinsic function) 350

__bis_GIE_interrupt_state (intrinsic function) 350
 __bis_SR_register (intrinsic function) 350
 __bis_SR_register_on_exit (intrinsic function) 350
 __BUILD_NUMBER__ (predefined symbol) 362
 __cc_version1 (extended keyword) 313
 choosing calling convention 179
 __cc_version2 (extended keyword) 314
 choosing calling convention 179
 __code_distance (intrinsic function) 350
 __CODE_MODEL__ (predefined symbol) 362
 __code, symbol used in library 377
 __constrange(), symbol used in library 377
 __construction_by_bitwise_copy_allowed, symbol used
 in library 377
 __CORE__ (predefined symbol) 362
 __COUNTER__ (predefined symbol) 362
 __cplusplus (predefined symbol) 362
 __DATA_MODEL__ (predefined symbol) 363
 __data16 (extended keyword) 301, 314
 __data16_read_addr (intrinsic function) 351
 __data16_size_t 208
 __data20 (extended keyword) 301, 314
 __data20_longjmp (intrinsic function) 352
 __data20_read_char (intrinsic function) 352
 __data20_read_long (intrinsic function) 352
 __data20_read_short (intrinsic function) 352
 __data20_setjmp (intrinsic function) 352
 __data20_write_char (intrinsic function) 353
 __data20_write_long (intrinsic function) 353
 __data20_write_short (intrinsic function) 353
 __DATE__ (predefined symbol) 363
 __delay_cycles (intrinsic function) 353
 __disable_interrupt (intrinsic function) 353
 __DLIB_FILE_DESCRIPTOR (configuration symbol) . . 147
 __embedded_cplusplus (predefined symbol) 363
 __enable_interrupt (intrinsic function) 354
 __even_in_range (intrinsic function) 75, 354
 __exit (library function) 137
 __FILE__ (predefined symbol) 363
 __FUNCTION__ (predefined symbol) 200, 364

<code>__func__</code> (predefined symbol)	200, 364	<code>__iar_pow_smallll</code> (library routine)	133
<code>__gets</code> , in <code>stdio.h</code>	376	<code>__iar_program_start</code> (label)	135
<code>__get_interrupt_state</code> (intrinsic function)	354	<code>__iar_Sin</code> (library routine)	133–134
<code>__get_R4_register</code> (intrinsic function)	355	<code>__iar_Sinf</code> (library routine)	133–134
<code>__get_R5_register</code> (intrinsic function)	355	<code>__iar_Sinl</code> (library routine)	133–134
<code>__get_SP_register</code> (intrinsic function)	355	<code>__iar_Sin_accurate</code> (library routine)	134
<code>__get_SR_register</code> (intrinsic function)	356	<code>__iar_sin_accurate</code> (library routine)	134
<code>__get_SR_register_on_exit</code> (intrinsic function)	356	<code>__iar_Sin_accuratef</code> (library routine)	134
<code>__has_constructor</code> , symbol used in library	377	<code>__iar_sin_accuratef</code> (library routine)	134
<code>__has_destructor</code> , symbol used in library	377	<code>__iar_Sin_accuratel</code> (library routine)	134
<code>__iar_cos_accurate</code> (library routine)	134	<code>__iar_sin_accuratel</code> (library routine)	134
<code>__iar_cos_accuratef</code> (library routine)	134	<code>__iar_Sin_small</code> (library routine)	133
<code>__iar_cos_accuratel</code> (library routine)	134	<code>__iar_sin_small</code> (library routine)	133
<code>__iar_cos_small</code> (library routine)	133	<code>__iar_Sin_smallf</code> (library routine)	133
<code>__iar_cos_smallf</code> (library routine)	133	<code>__iar_sin_smallf</code> (library routine)	133
<code>__iar_cos_smallll</code> (library routine)	133	<code>__iar_Sin_smallll</code> (library routine)	133
<code>__IAR_DLIB_PERTHREAD_INIT_SIZE</code> (macro)	156	<code>__iar_sin_smallll</code> (library routine)	133
<code>__IAR_DLIB_PERTHREAD_SIZE</code> (macro)	156	<code>__IAR_SYSTEMS_ICC__</code> (predefined symbol)	364
<code>__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET</code> (symbolptr)	156	<code>__iar_tan_accurate</code> (library routine)	134
<code>__iar_exp_small</code> (library routine)	133	<code>__iar_tan_accuratef</code> (library routine)	134
<code>__iar_exp_smallf</code> (library routine)	133	<code>__iar_tan_accuratel</code> (library routine)	134
<code>__iar_exp_smallll</code> (library routine)	133	<code>__iar_tan_small</code> (library routine)	133
<code>__iar_log_small</code> (library routine)	133	<code>__iar_tan_smallf</code> (library routine)	133
<code>__iar_log_smallf</code> (library routine)	133	<code>__iar_tan_smallll</code> (library routine)	133
<code>__iar_log_smallll</code> (library routine)	133	<code>__interrupt</code> (extended keyword)	74, 315
<code>__iar_log10_small</code> (library routine)	133	using in pragma directives	346
<code>__iar_log10_smallf</code> (library routine)	133	<code>__intrinsic</code> (extended keyword)	315
<code>__iar_log10_smallll</code> (library routine)	133	<code>__LINE__</code> (predefined symbol)	364
<code>__iar_Pow</code> (library routine)	134	<code>__low_level_init</code>	136
<code>__iar_Powf</code> (library routine)	134	initialization phase	50
<code>__iar_Powl</code> (library routine)	134	<code>__low_level_init</code> , customizing	138
<code>__iar_Pow_accurate</code> (library routine)	134	<code>__low_power_mode_n</code> (intrinsic function)	356
<code>__iar_pow_accurate</code> (library routine)	134	<code>__low_power_mode_off_on_exit</code> (intrinsic function)	356
<code>__iar_Pow_accuratef</code> (library routine)	134	<code>__memory_of</code> operator	205
<code>__iar_pow_accuratef</code> (library routine)	134	symbol used in library	377
<code>__iar_Pow_accuratel</code> (library routine)	134	<code>__monitor</code> (extended keyword)	315
<code>__iar_pow_accuratel</code> (library routine)	134	<code>__noreturn</code> (extended keyword)	318
<code>__iar_pow_small</code> (library routine)	133	<code>__no_alloc</code> (extended keyword)	316
<code>__iar_pow_smallf</code> (library routine)	133		

- `__no_alloc_str` (operator) 316
- `__no_alloc_str16` (operator) 316
- `__no_alloc16` (extended keyword) 316
- `__no_init` (extended keyword) 245, 317
- `__no_multiplier` (extended keyword) 317
- `__no_operation` (intrinsic function) 356
- `__no_pic` (extended keyword) 318
- `__op_code` (intrinsic function) 357
- `__persistent` (extended keyword) 318
 - cannot be used with ROPI 80
- `__POSITION_INDEPENDENT_CODE__` (predefined symbol) 364
- `__PRETTY_FUNCTION__` (predefined symbol) 365
- `__printf_args` (pragma directive) 340
- `__program_start` (label) 135
- `__ramfunc` (extended keyword) 79, 319
 - cannot be used with ROPI 80
- `__raw` (extended keyword) 319
- `__REGISTER_MODEL__` (predefined symbol) 365
- `__REGISTER_R4__` (predefined symbol) 365
- `__REGISTER_R5__` (predefined symbol) 365
- `__ReportAssert` (library function) 144
- `__root` (extended keyword) 320
- `__ROPI__` (predefined symbol) 365
- `__ro_placement` (extended keyword) 320
- `__rt_version` (runtime model attribute) 113
- `__saturated_add_signed_type` (intrinsic function) 357
- `__saturated_add_unsigned_type` (intrinsic function) 357
- `__saturated_sub_signed_type` (intrinsic function) 357
- `__saturated_sub_unsigned_type` (intrinsic function) 358
- `__scanf_args` (pragma directive) 341
- `__segment_begin` (extended operator) 197
- `__segment_end` (extended operator) 197
- `__segment_size` (extended operator) 197
- `__set_interrupt_state` (intrinsic function) 358
- `__set_R4_register` (intrinsic function) 358
- `__set_R5_register` (intrinsic function) 358
- `__set_SP_register` (intrinsic function) 359
- `__STDC_VERSION__` (predefined symbol) 366
- `__STDC__` (predefined symbol) 365
- `__swap_bytes` (intrinsic function) 359
- `__task` (extended keyword) 321
- `__TIMESTAMP__` (predefined symbol) 366
- `__TIME__` (predefined symbol) 366
- `__ungetchar`, in `stdio.h` 376
- `__VA_ARGS__` (preprocessor extension) 193
- `__write_array`, in `stdio.h` 376
- `__write_buffered` (DLIB library function) 121
- `-D` (compiler option) 263
- `-e` (compiler option) 270
- `-f` (compiler option) 272
- `-I` (compiler option) 274
- `-l` (compiler option) 274
 - for creating skeleton code 177
- `-O` (compiler option) 283
- `-o` (compiler option) 284
- `-Q` (linker option) 108
- `-r` (compiler option) 265
- `--char_is_signed` (compiler option) 262
- `--char_is_unsigned` (compiler option) 262
- `--clib` (compiler option) 262
- `--code_model` (compiler option) 262
- `--c89` (compiler option) 261
- `--data_model` (compiler option) 264
- `--debug` (compiler option) 265
- `--dependencies` (compiler option) 265
- `--diagnostics_tables` (compiler option) 268
- `--diag_error` (compiler option) 266
- `--diag_remark` (compiler option) 266
- `--diag_suppress` (compiler option) 267
- `--diag_warning` (compiler option) 267
- `--discard_unused_publics` (compiler option) 268
- `--dlib` (compiler option) 268
- `--dlib_config` (compiler option) 269
- `--double` (compiler option) 270
- `--ec++` (compiler option) 270
- `--eec++` (compiler option) 271
- `--enable_multibytes` (compiler option) 271
- `--enable_restrict` (compiler option) 271

--error_limit (compiler option)	272
--guard_calls (compiler option).	272
--header_context (compiler option).	273
--hw_workaround (compiler option).	273
--library_module (compiler option)	275
--lock_r4 (compiler option).	275
--lock_r5 (compiler option).	275
--macro_positions_in_diagnostics (compiler option)	276
--mfc (compiler option).	276
--migration_preprocessor_extensions (compiler option).	276
--misrac (compiler option)	259
--misrac_verbose (compiler option)	259
--misrac1998 (compiler option)	259
--misrac2004 (compiler option)	259
--module_name (compiler option)	277
--multiplier (compiler option)	277
--no_call_frame_info (compiler option)	278
--no_code_motion (compiler option)	278
--no_cse (compiler option)	278
--no_inline (compiler option)	279
--no_path_in_file_macros (compiler option).	279
--no_rw_dynamic_init (compiler option)	279
--no_size_constraints (compiler option)	280
--no_static_destruction (compiler option)	280
--no_system_include (compiler option)	280
--no_typedefs_in_diagnostics (compiler option).	281
--no_ubrof_messages (compiler option)	282
--no_unroll (compiler option)	282
--no_warnings (compiler option)	282
--no_wrap_diagnostics (compiler option)	282
--omit_types (compiler option)	283
--only_stdout (compiler option)	284
--output (compiler option).	284
--pending_instantiations (compiler option).	284
--predef_macro (compiler option).	285
--preinclude (compiler option)	285
--preprocess (compiler option)	285
--reduce_stack_usage (compiler option)	286
--regvar_r4 (compiler option)	286
--regvar_r5 (compiler option)	287
--relaxed_fp (compiler option)	287
--remarks (compiler option)	288
--require_prototypes (compiler option).	288
--ropi (compiler option)	288
--save_reg20 (compiler option).	289
--segment (compiler option)	289
--silent (compiler option)	290
--strict (compiler option).	290
--system_include_dir (compiler option)	291
--use_c++_inline (compiler option)	291
--version (compiler option)	291
--vla (compiler option)	292
--warnings_affect_exit_code (compiler option)	252, 292
--warnings_are_errors (compiler option)	292
--warn_about_c_style_casts (compiler option)	292
?C_EXIT (assembler label).	165
?C_GETCHAR (assembler label)	165
?C_PUTCHAR (assembler label)	165
@ (operator)	
placing at absolute address.	231
placing in segments	233
#include files, specifying	250, 274
#warning message (preprocessor extension)	367
%Z replacement string,	
implementation-defined behavior	418

Numerics

32-bits (floating-point format)	300
64-bit data types, avoiding	227
64-bits (floating-point format)	300