



**IAR Embedded
Workbench**

IAR C/C++ Development Guide

Compiling and linking

for the Renesas

RL78 Microcontroller Family

COPYRIGHT NOTICE

© 2011–2019 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, Embedded Trust, C-Trust, IAR Connect, C-SPY, C-RUN, C-STAT, IAR Visual State, IAR KickStart Kit, 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.

Renesas is a registered trademark of Renesas Electronics Corporation. RL78 is a trademark of Renesas Electronics 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

Fifth edition: November 2019

Part number: DRL78_I-5

This guide applies to version 4.x of IAR Embedded Workbench® for the Renesas RL78 microcontroller family.

The *IAR C/C++ Development Guide for RL78* replaces all versions of the *IAR C/C++ Compiler Reference Guide for RL78* and the *IAR Linker and Library Tools Reference Guide*.

Internal reference: BB5, csrct2010.1, V_110411, IJOA.

Brief contents

Tables	33
Preface	35
Part 1. Using the build tools	43
Introduction to the IAR build tools	45
Developing embedded applications	51
Data storage	65
Functions	77
Linking using ILINK	87
Linking your application	105
The DLIB runtime environment	121
Assembler language interface	159
Using C	183
Using C++	191
Application-related considerations	205
Efficient coding for embedded applications	219
Part 2. Reference information	237
External interface details	239
Compiler options	249
Linker options	291
Data representation	321
Extended keywords	335

Pragma directives	353
Intrinsic functions	381
The preprocessor	385
C/C++ standard library functions	393
The linker configuration file	405
Section reference	437
The stack usage control file	449
IAR utilities	457
Implementation-defined behavior for Standard C++	503
Implementation-defined behavior for Standard C	523
Implementation-defined behavior for C89	543
Index	555

Contents

Tables	33
Preface	35
Who should read this guide	35
Required knowledge	35
How to use this guide	35
What this guide contains	36
Part 1. Using the build tools	36
Part 2. Reference information	36
Other documentation	37
User and reference guides	38
The online help system	38
Further reading	39
Web sites	39
Document conventions	39
Typographic conventions	40
Naming conventions	41
Part I. Using the build tools	43
Introduction to the IAR build tools	45
The IAR build tools—an overview	45
The IAR C/C++ Compiler	45
The IAR Assembler	46
The IAR ILINK Linker	46
Specific ELF tools	46
External tools	46
IAR language overview	47
Device support	47
Supported RL78 devices	47
Preconfigured support files	48
Examples for getting started	48

Special support for embedded systems	48
Extended keywords	49
Pragma directives	49
Predefined symbols	49
Accessing low-level features	49
Developing embedded applications	51
Developing embedded software using IAR build tools	51
CPU features and constraints	51
Mapping of memory	51
Communication with peripheral units	52
Event handling	52
System startup	52
Real-time operating systems	53
Interoperability with other build tools	53
The build process—an overview	54
The translation process	54
The linking process	55
After linking	56
Application execution—an overview	56
The initialization phase	57
The execution phase	60
The termination phase	60
Building applications—an overview	61
Basic project configuration	61
Processor core	62
Data model	62
Code model	63
Size of double floating-point type	63
Optimization for speed and size	63
Data storage	65
Introduction	65
Different ways to store data	65

Memory types	66
Introduction to memory types	66
Using data memory attributes	67
Pointers and memory types	69
Structures and memory types	70
More examples	70
C++ and memory types	71
Data models	72
Specifying a data model	72
Storage of auto variables and parameters	73
The stack	73
Short address working area	74
Dynamic memory on the heap	75
Potential problems	75
Functions	77
Function-related extensions	77
Code models and memory attributes for function storage	77
Using function memory attributes	78
Primitives for interrupts, concurrency, and OS-related programming	79
Interrupt functions	79
Monitor functions	80
Inlining functions	83
C versus C++ semantics	84
Features controlling function inlining	84
Stack protection	85
Stack protection in the IAR C/C++ Compiler	85
Using stack protection in your application	86
Linking using ILINK	87
Linking—an overview	87
Modules and sections	88
The linking process in detail	89

Placing code and data—the linker configuration file	91
A simple example of a configuration file	92
Initialization at system startup	94
The initialization process	95
C++ dynamic initialization	96
Additional compiler-generated sections	96
Stack usage analysis	97
Introduction to stack usage analysis	97
Performing a stack usage analysis	97
Result of an analysis—the map file contents	98
Specifying additional stack usage information	100
Limitations	101
Situations where warnings are issued	102
Call graph log	102
Call graph XML output	103
Linking your application	105
Linking considerations	105
Choosing a linker configuration file	105
Defining your own memory areas	106
Placing sections	107
Reserving space in RAM	108
Keeping modules	109
Keeping symbols and sections	109
Application startup	109
Setting up stack memory	109
Setting up heap memory	110
Setting up the atexit limit	110
Changing the default initialization	110
Interaction between ILINK and the application	114
Standard library handling	115
Producing other output formats than ELF/DWARF	115
Linking object files produced by the Renesas CC-RL compiler	115

Hints for troubleshooting	115
Relocation errors	116
Checking module consistency	117
Runtime model attributes	118
Using runtime model attributes	118
Linker optimizations	119
Virtual function elimination	119
Small function inlining	120
Duplicate section merging	120
The DLIB runtime environment	121
Introduction to the runtime environment	121
Runtime environment functionality	121
Briefly about input and output (I/O)	122
Briefly about C-SPY emulated I/O	123
Briefly about retargeting	124
Setting up the runtime environment	125
Setting up your runtime environment	125
Retargeting—Adapting for your target system	127
Overriding library modules	128
Customizing and building your own runtime library	129
Additional information on the runtime environment	131
Bounds checking functionality	131
Runtime library configurations	131
Prebuilt runtime libraries	132
Formatters for printf	136
Formatters for scanf	137
The C-SPY emulated I/O mechanism	138
Math functions	139
System startup and termination	141
System initialization	144
The DLIB low-level I/O interface	145
abort	146
clock	146

__close	147
__exit	147
getenv	147
__getzone	148
__iar_ReportAssert	149
__lseek	149
__open	149
raise	150
__read	150
remove	151
rename	152
signal	152
system	152
__time32, __time64	153
__write	153
Configuration symbols for file input and output	154
Locale	155
Strtod	156
Hardware support	156
Managing a multithreaded environment	157
Multithread support in the DLIB runtime environment	157
Enabling multithread support	158
Assembler language interface	159
Mixing C and assembler	159
Intrinsic functions	159
Mixing C and assembler modules	160
Inline assembler	161
Calling assembler routines from C	163
Creating skeleton code	163
Compiling the skeleton code	164
Calling assembler routines from C++	165
Calling convention	166
Choosing a calling convention	166

Function declarations	167
Using C linkage in C++ source code	167
Registers and bank switching	167
Preserved versus scratch registers	168
Function entrance	168
Function exit	170
Bitfield allocation	171
Examples	171
Assembler instructions used for calling functions	173
Calling functions in the Near code model	173
Calling functions in the Far code model	174
Making function table calls	174
Memory access methods	174
The saddr memory access method	175
The near memory access method	175
The far memory access method	176
The huge access method	176
Call frame information	177
CFI directives	177
Creating assembler source with CFI support	178
Using C	183
C language overview	183
Extensions overview	183
Enabling language extensions	185
IAR C language extensions	185
Extensions for embedded systems programming	185
Relaxations to Standard C	187
Using C++	191
Overview—Standard C++	191
Exceptions and RTTI	191
Enabling support for C++	192
C++ feature descriptions	192
Using IAR attributes with classes	192

Templates	195
auto and decltype	197
Function types	197
New and Delete operators	198
Using static class objects in interrupts	199
Using New handlers	200
Pointer to member functions	200
Debug support in C-SPY	200
C++ language extensions	201
Porting code from EC++ or EEC++	203
Application-related considerations	205
Output format considerations	205
Stack considerations	206
Stack size considerations	206
Heap considerations	206
Heap sections in DLIB	206
Heap size and standard I/O	207
Interaction between the tools and your application	207
Checksum calculation for verifying image integrity	209
Briefly about checksum calculation	209
Calculating and verifying a checksum	211
Troubleshooting checksum calculation	216
Patching symbol definitions using \$Super\$\$ and \$Sub\$\$...	218
An example using the \$Super\$\$ and \$Sub\$\$ patterns	218
Efficient coding for embedded applications	219
Selecting data types	219
Using efficient data types	219
Floating-point types	219
Alignment of elements in a structure	220
Anonymous structs and unions	220
Controlling data and function placement in memory	222
Data placement at an absolute location	223
Data and function placement in sections	224

Controlling compiler optimizations	226
Scope for performed optimizations	226
Multi-file compilation units	226
Optimization levels	227
Speed versus size	228
Fine-tuning enabled transformations	228
Facilitating good code generation	231
Writing optimization-friendly source code	232
Saving stack space and RAM memory	232
Function prototypes	232
Integer types and bit negation	233
Protecting simultaneously accessed variables	234
Accessing special function registers	234
Non-initialized variables	235
Part 2. Reference information	237
External interface details	239
Invocation syntax	239
Compiler invocation syntax	239
ILINK invocation syntax	240
Passing options	240
Environment variables	241
Include file search procedure	241
Compiler output	242
Error return codes	243
ILINK output	244
Text encodings	244
Characters and string literals	245
Reserved identifiers	246
Diagnostics	246
Message format for the compiler	246
Message format for the linker	247
Severity levels	247

Setting the severity level	248
Internal error	248
Compiler options	249
Options syntax	249
Types of options	249
Rules for specifying parameters	249
Summary of compiler options	251
Descriptions of compiler options	256
--c89	256
--calling_convention	256
--char_is_signed	257
--char_is_unsigned	257
--code_model	257
--code_section	258
--core	258
--c++	259
-D	259
--data_model	260
--debug, -r	260
--dependencies	260
--deprecated_feature_warnings	262
--diag_error	262
--diag_remark	263
--diag_suppress	263
--diag_warning	263
--diagnostics_tables	264
--disable_div_mod_instructions	264
--discard_unused_publics	264
--dlib_config	265
--do_explicit_zero_opt_in_named_sections	266
--double	266
-e	267
--enable_restrict	267

--error_limit	267
-f	268
--f	268
--generate_callt_runtime_library_calls	269
--generate_far_runtime_library_calls	269
--guard_calls	269
--header_context	270
-I	270
-l	270
--macro_positions_in_diagnostics	271
--max_cost_constexpr_call	272
--max_depth_constexpr_call	272
--mfc	272
--near_const_location	273
--no_bom	273
--no_call_frame_info	274
--no_clustering	274
--no_code_motion	274
--no_cross_call	275
--no_cse	275
--no_fragments	275
--no_inline	276
--no_path_in_file_macros	276
--no_scheduling	276
--no_size_constraints	276
--no_static_destruction	277
--no_system_include	277
--no_tbaa	277
--no_typedefs_in_diagnostics	278
--no_uniform_attribute_syntax	278
--no_unroll	279
--no_warnings	279
--no_wrap_diagnostics	279
--nonportable_path_warnings	279

-O	280
--only_stdout	280
--output, -o	281
--pending_instantiations	281
--predef_macros	281
--preinclude	282
--preprocess	282
--public_equ	282
--relaxed_fp	283
--remarks	284
--require_prototypes	284
--silent	284
--source_encoding	285
--stack_protection	285
--strict	285
--system_include_dir	286
--text_out	286
--uniform_attribute_syntax	287
--use_cplusplus_inline	287
--use_paths_as_written	287
--use_unix_directory_separators	288
--utf8_text_in	288
--version	288
--vla	289
--warn_about_c_style_casts	289
--warnings_affect_exit_code	289
--warnings_are_errors	289
--workseg_area	290
Linker options	291
Summary of linker options	291
Descriptions of linker options	294
--call_graph	294
--config	295

--config_def	295
--config_search	295
--cpp_init_routine	296
--debug_lib	296
--default_to_complex_ranges	297
--define_symbol	297
--dependencies	297
--diag_error	298
--diag_remark	299
--diag_suppress	299
--diag_warning	300
--diagnostics_tables	300
--enable_stack_usage	300
--entry	301
--entry_list_in_address_order	301
--error_limit	301
--export_builtin_config	302
-f	302
--f	302
--force_output	303
--image_input	303
--inline	304
--keep	304
--log	305
--log_file	306
--mangled_names_in_messages	306
--manual_dynamic_initialization	306
--map	307
--merge_duplicate_sections	307
--no_bom	308
--no_entry	308
--no_fragments	308
--no_free_heap	309
--no_inline	309

--no_library_search	309
--no_locals	310
--no_range_reservations	310
--no_remove	310
--no_vfe	310
--no_warnings	311
--no_wrap_diagnostics	311
--only_stdout	311
--output, -o	312
--place_holder	312
--preconfig	313
--printf_multibytes	313
--redirect	313
--remarks	314
--scanf_multibytes	314
--search, -L	314
--silent	315
--stack_usage_control	315
--strip	315
--text_out	316
--threaded_lib	316
--timezone_lib	316
--use_full_std_template_names	317
--utf8_text_in	317
--version	317
--vfe	318
--warnings_affect_exit_code	318
--warnings_are_errors	318
--whole_archive	319
Data representation	321
Alignment	321
Alignment on the RL78 microcontroller	322

Basic data types—integer types	322
Integer types—an overview	322
Bool	323
The enum type	323
The char type	323
The wchar_t type	324
The char16_t type	324
The char32_t type	324
Bitfields	324
Basic data types—floating-point types	326
Floating-point environment	327
32-bit floating-point format	327
64-bit floating-point format	327
Representation of special floating-point numbers	328
Pointer types	328
Function pointers	328
Data pointers	329
Casting	329
Structure types	330
Alignment of structure types	330
General layout	330
Packed structure types	331
Type qualifiers	332
Declaring objects volatile	332
Declaring objects volatile and const	333
Declaring objects const	334
Data types in C++	334
Extended keywords	335
General syntax rules for extended keywords	335
Type attributes	335
Object attributes	338
Summary of extended keywords	339

Descriptions of extended keywords	340
__callt	340
__far	341
__far_func	341
__huge	342
__interrupt	342
__intrinsic	343
__monitor	343
__near	343
__near_func	344
__no_alloc, __no_alloc16	344
__no_alloc_str, __no_alloc_str16	344
__no_bit_access	345
__no_init	346
__noreturn	346
__no_save	346
__packed	347
__root	348
__ro_placement	348
__saddr	349
__sfr	349
__v1_call	350
__v2_call	350
__weak	351
Supported GCC attributes	351
Pragma directives	353
Summary of pragma directives	353
Descriptions of pragma directives	355
bank	355
basic_template_matching	356
bitfields	356
calls	357
call_graph_root	358

constseg	358
data_alignment	359
dataseg	359
default_function_attributes	360
default_variable_attributes	361
deprecated	362
diag_default	363
diag_error	363
diag_remark	364
diag_suppress	364
diag_warning	364
error	365
function_category	365
include_alias	366
inline	366
language	367
location	368
message	368
no_stack_protect	369
no_workseg	369
object_attribute	369
optimize	370
pack	371
__printf_args	372
public_equ	372
required	373
rtmodel	373
__scanf_args	374
section	374
stack_protect	375
STDC CX_LIMITED_RANGE	375
STDC FENV_ACCESS	376
STDC FP_CONTRACT	376
type_attribute	377

unroll	377
vector	378
weak	378
Intrinsic functions	381
Summary of intrinsic functions	381
Descriptions of intrinsic functions	382
__break	382
__disable_interrupt	382
__enable_interrupt	382
__get_interrupt_level	382
__get_interrupt_state	382
__halt	383
__mach	383
__machu	383
__no_operation	383
__rollb	383
__rollw	384
__ror1b	384
__ror1w	384
__set_interrupt_level	384
__set_interrupt_state	384
__stop	384
The preprocessor	385
Overview of the preprocessor	385
Description of predefined preprocessor symbols	386
__BASE_FILE__	386
__BUILD_NUMBER__	386
__CALLING_CONVENTION__	386
__CODE_MODEL__	386
__CORE__	386
__COUNTER__	387
__cplusplus	387
__DATA_MODEL__	387

__DATE__	387
__EXCEPTIONS__	387
__FAR_RUNTIME_ATTRIBUTE__	387
__FILE__	388
__func__	388
__FUNCTION__	388
__IAR_SYSTEMS_ICC__	388
__ICC r178 __	388
__LINE__	389
__LITTLE_ENDIAN__	389
__PRETTY_FUNCTION__	389
__RTTI__	389
__STDC__	389
__STDC_LIB_EXT1__	389
__STDC_NO_ATOMICS__	390
__STDC_NO_THREADS__	390
__STDC_NO_VLA__	390
__STDC_UTF16__	390
__STDC_UTF32__	390
__STDC_VERSION__	390
__SUBVERSION__	390
__TIME__	391
__TIMESTAMP__	391
__VER__	391
Descriptions of miscellaneous preprocessor extensions	391
NDEBUG	391
__STDC_WANT_LIB_EXT1__	392
#warning message	392
C/C++ standard library functions	393
C/C++ standard library overview	393
Header files	393
Library object files	394
Alternative more accurate library functions	394

Reentrancy	394
The longjmp function	395
DLIB runtime environment—implementation details	395
Briefly about the DLIB runtime environment	395
C header files	396
C++ header files	397
Library functions as intrinsic functions	401
Not supported C/C++ functionality	401
Atomic operations	401
Added C functionality	401
Non-standard implementations	404
Symbols used internally by the library	404
The linker configuration file	405
Overview	405
Declaring the build type	406
build for directive	407
Defining memories and regions	407
define memory directive	408
define region directive	408
logical directive	409
Regions	411
Region literal	411
Region expression	412
Empty region	413
Section handling	414
define block directive	415
define overlay directive	417
initialize directive	418
do not initialize directive	421
keep directive	421
place at directive	422
place in directive	423
use init format directive	425

use init table directive	425
Section selection	426
section-selectors	427
extended-selectors	430
Using symbols, expressions, and numbers	431
check that directive	431
define symbol directive	432
export directive	433
expressions	433
numbers	434
Structural configuration	435
error directive	435
if directive	436
include directive	436
Section reference	437
Summary of sections	437
Descriptions of sections and blocks	439
.bss	439
.bss.noinit	439
.bssf	439
.bssf.noinit	440
.callt0	440
.const	440
.constf	440
.consth	441
CSTACK	441
.data	441
.data_init	441
.dataf	442
.dataf_init	442
FAR_HEAP	442
.hbss	442
.hbss.noinit	442

.hdata	443
.hdata_init	443
HUGE_HEAP	443
.iar.dynexit	443
__iar_tls.\$DATA	444
.init_array	444
.intvec	444
NEAR_HEAP	444
.option_byte	444
.preinit_array	445
.sbss	445
.sbss.noinit	445
.sdata	445
.sdata_init	446
.security_id	446
.switch	446
.switchf	446
.text	446
.textf	447
.wrkseg	447

The stack usage control file	449
------------------------------------	-----

Overview	449
-----------------------	-----

C++ names	449
-----------------	-----

Stack usage control directives	450
---	-----

call graph root directive	450
---------------------------------	-----

exclude directive	450
-------------------------	-----

function directive	450
--------------------------	-----

max recursion depth directive	451
-------------------------------------	-----

no calls from directive	452
-------------------------------	-----

possible calls directive	452
--------------------------------	-----

Syntactic components	453
-----------------------------------	-----

<i>category</i>	453
-----------------------	-----

<i>func-spec</i>	453
------------------------	-----

<i>module-spec</i>	453
<i>name</i>	454
<i>call-info</i>	454
<i>stack-size</i>	454
<i>size</i>	455
IAR utilities	457
The IAR Archive Tool—iarchive	457
Invocation syntax	457
Summary of iarchive commands	458
Summary of iarchive options	459
Diagnostic messages	459
The IAR ELF Tool—ielftool	460
Invocation syntax	461
Summary of ielftool options	461
The IAR ELF Dumper—ielfdump	462
Invocation syntax	463
Summary of ielfdump options	463
The IAR ELF Object Tool—iobjmanip	464
Invocation syntax	464
Summary of iobjmanip options	465
Diagnostic messages	465
The IAR Absolute Symbol Exporter—ismexport	467
Invocation syntax	467
Summary of ismexport options	468
Steering files	469
Hide directive	469
Rename directive	470
Show directive	470
Show-root directive	471
Show-weak directive	471
Diagnostic messages	471
Descriptions of options	473
--a	473

--all	474
--bin	474
--bin-multi	475
--checksum	475
--code	480
--create	480
--delete, -d	480
--disasm_data	481
--edit	481
--export_locals	481
--extract, -x	482
-f	482
--f	483
--fill	484
--front_headers	484
--generate_vfe_header	485
--ihex	485
--no_bom	485
--no_header	486
--no_rel_section	486
--no_strtab	486
--no_utf8_in	487
--offset	487
--output, -o	488
--parity	488
--ram_reserve_ranges	489
--range	490
--raw	490
--remove_file_path	491
--remove_section	491
--rename_section	492
--rename_symbol	492
--replace, -r	492
--reserve_ranges	493

--section, -s	494
--segment, -g	494
--self_reloc	495
--show_entry_as	495
--silent	495
--simple	496
--simple-ne	496
--source	496
--srec	497
--srec-len	497
--srec-s3only	497
--strip	498
--symbols	498
--text_out	498
--titxt	499
--toc, -t	499
--use_full_std_template_names	500
--utf8_text_in	500
--verbose, -V	500
--version	501
--vtoc	501
Implementation-defined behavior for Standard C++	503
Descriptions of implementation-defined behavior for C++	503
1 General	503
2 Lexical conventions	504
3 Basic concepts	506
4 Standard conversions	508
5 Expressions	509
7 Declarations	510
8 Declarators	510
9 Classes	511
14 Templates	511
15 Exception handling	511

16 Preprocessing directives	511
17 Library introduction	512
18 Language support library	513
20 General utilities library	514
21 Strings library	515
22 Localization library	516
23 Containers library	517
25 Algorithms library	517
27 Input/output library	517
28 Regular expressions library	518
29 Atomic operations library	519
30 Thread support library	519
Annex D (normative): Compatibility features	519
Implementation quantities	519
Implementation-defined behavior for Standard C	523
Descriptions of implementation-defined behavior	523
J.3.1 Translation	523
J.3.2 Environment	524
J.3.3 Identifiers	525
J.3.4 Characters	525
J.3.5 Integers	527
J.3.6 Floating point	528
J.3.7 Arrays and pointers	529
J.3.8 Hints	529
J.3.9 Structures, unions, enumerations, and bitfields	529
J.3.10 Qualifiers	530
J.3.11 Preprocessing directives	530
J.3.12 Library functions	533
J.3.13 Architecture	538
J.4 Locale	539
Implementation-defined behavior for C89	543
Descriptions of implementation-defined behavior	543
Translation	543

Environment	543
Identifiers	544
Characters	544
Integers	545
Floating point	546
Arrays and pointers	547
Registers	547
Structures, unions, enumerations, and bitfields	547
Qualifiers	548
Declarators	548
Statements	548
Preprocessing directives	548
Library functions for the IAR DLIB runtime environment	550
Index	555

Tables

1: Typographic conventions used in this guide	40
2: Naming conventions used in this guide	41
3: Processor cores	62
4: Memory types and their corresponding keywords	67
5: Data model characteristics	72
6: Heaps supported in memory types	75
7: Code models	78
8: Function memory attributes	78
9: Sections holding initialized data	94
10: Description of a relocation error	116
11: Example of runtime model attributes	118
12: Library configurations	132
13: Formatters for printf	136
14: Formatters for scanf	137
15: Library objects using TLS	157
16: Registers used for passing parameters	169
17: Registers used for returning values	170
18: Call frame information resources defined in a names block	178
19: Language extensions	185
20: Section operators and their symbols	187
21: Compiler optimization levels	227
22: Compiler environment variables	241
23: ILINK environment variables	241
24: Error return codes	243
25: Compiler options summary	251
26: Linker options summary	291
27: Integer types	322
28: Floating-point types	326
29: Function pointers	328
30: Data pointers	329
31: Extended keywords summary	339

32: Pragma directives summary	353
33: Intrinsic functions summary	381
34: Traditional Standard C header files—DLIB	396
35: C++ header files	397
36: New Standard C header files—DLIB	400
37: Examples of section selector specifications	429
38: Section summary	437
39: iarchive parameters	458
40: iarchive commands summary	458
41: iarchive options summary	459
42: ielftool parameters	461
43: ielftool options summary	461
44: ielfdumprl78 parameters	463
45: ielfdumprl78 options summary	463
46: iobjmanip parameters	464
47: iobjmanip options summary	465
48: isymexport parameters	467
49: isymexport options summary	468
50: Execution character sets and their encodings	504
51: C++ implementation quantities	519
52: Execution character sets and their encodings	526
53: Translation of multibyte characters in the extended source character set	539
54: Message returned by strerror()—DLIB runtime environment	541
55: Execution character sets and their encodings	544
56: Message returned by strerror()—DLIB runtime environment	553

Preface

Welcome to the *IAR C/C++ Development Guide for RL78*. The purpose of this guide is to provide you with detailed reference information that can help you to use the build tools 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 RL78 microcontroller, and need detailed reference information on how to use the build tools.

REQUIRED KNOWLEDGE

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

- The architecture and instruction set of the RL78 microcontroller family (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 37.

How to use this guide

When you start using the IAR C/C++ compiler and linker for RL78, you should read *Part 1. Using the build tools* in this guide.

When you are familiar with the compiler and linker, 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 go through the tutorials, which you can find in IAR Information Center in the product. They 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 BUILD TOOLS

- *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 RL78 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 using ILINK* describes the linking process using the IAR ILINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using ILINK 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.s`, how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two 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 level of C++ support.
- *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 and linker interact with their environment—the invocation syntax, methods for passing options to the compiler and linker, environment variables, the include file

search procedure, and the different types of compiler and linker output. The chapter also describes how the 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.
- *Linker options* gives a summary of the options, and contains detailed reference information for each linker 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 RL78-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 RL78-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.
- *The linker configuration file* describes the purpose of the linker configuration file, and describes its contents.
- *Section reference* gives reference information about the use of sections.
- *The stack usage control file* describes the syntax and semantics of stack usage control files.
- *IAR utilities* describes the IAR utilities that handle the ELF and DWARF object formats.
- *Implementation-defined behavior for Standard C++* describes how the compiler handles the implementation-defined areas of Standard C++.
- *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 are available in the *Installation and Licensing Quick Reference Guide* and the *Licensing Guide*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for RL78*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for RL78*.
- Programming for the IAR C/C++ Compiler for RL78 and linking using the IAR ILINK Linker, is available in the *IAR C/C++ Development Guide for RL78*.
- Programming for the IAR Assembler for RL78, is available in the *IAR Assembler User Guide for RL78*.
- 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*.
- Migrating from an older UBROF-based product version to a newer version that uses the ELF/DWARF object format, is available in the guide *IAR Embedded Workbench® Migrating from UBROF to ELF/DWARF*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains information about:

- IDE project management and building
- Debugging using the IAR C-SPY® Debugger
- The IAR C/C++ Compiler
- The IAR Assembler
- 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.
- C-STAT
- MISRA C

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.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++*. Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.
- Meyers, Scott. *Effective STL*. Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.

The web site isocpp.org also has a list of recommended books about C++ programming.

WEB SITES

Recommended web sites:

- The Renesas web site, www.renesas.com, that contains information and news about the RL78 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.
- The C++ programming language web site, isocpp.org. This web site also has a list of recommended books about C++ programming.
- The C and C++ reference web site, en.cppreference.com.

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 `r178\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\r178\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
computer	<ul style="list-style-type: none"> • Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a linker or stack usage control directive, where [and] are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
{option}	A mandatory part of a linker or stack usage control directive, where { and } are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
[option]	An optional part of a command line option, pragma directive, or library filename.
[a b c]	An optional part of a command line option, pragma directive, or library filename with alternatives.
{a b c}	A mandatory part of a command line option, pragma directive, or library filename with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> • A cross-reference within this guide or to another guide. • Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

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

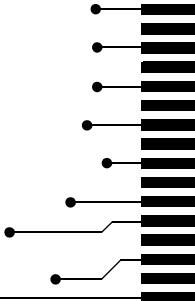
Brand name	Generic term
IAR Embedded Workbench® for RL78	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for RL78	the IDE
IAR C-SPY® Debugger for RL78	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for RL78	the compiler
IAR Assembler™ for RL78	the assembler
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Runtime Environment™	the DLIB runtime environment

Table 2: Naming conventions used in this guide

Part I. Using the build tools

This part of the *IAR C/C++ Development Guide for RL78* includes these chapters:

- Introduction to the IAR build tools
- Developing embedded applications
- Data storage
- Functions
- Linking using ILINK
- Linking your application
- The DLIB 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 RL78-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.



IAR Embedded Workbench® is a 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, and comprehensive and specific target support. IAR Embedded Workbench promotes a useful working methodology, and therefore a significant reduction in development time.

For information about the IDE, see the *IDE Project Management and Building Guide for RL78*.



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.

THE IAR C/C++ COMPILER

The IAR C/C++ Compiler for RL78 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 RL78-specific facilities.

THE IAR ASSEMBLER

The IAR Assembler for RL78 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 RL78 uses the same mnemonics and operand syntax as the Renesas RL78 Assembler, which simplifies the migration of existing code. For more information, see the *IAR Assembler User Guide for RL78*.

THE IAR ILINK LINKER

The IAR ILINK Linker for RL78 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.

SPECIFIC ELF TOOLS

ILINK both uses and produces industry-standard ELF and DWARF as object format, additional IAR utilities that handle these formats are provided:

- The IAR Archive Tool—`iarchive`—creates and manipulates a library (archive) of several ELF object files
- The IAR ELF Tool—`ielftool`—performs various transformations on an ELF executable image (such as, fill, checksum, format conversion etc)
- The IAR ELF Dumper for RL78—`ielfdumprl78`—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—`iobjmanip`—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—`isymexport`—exports absolute symbols from a ROM image file, so that they can be used when linking an add-on application.

EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IDE Project Management and Building Guide for RL78*.

IAR language overview

The IAR C/C++ Compiler for RL78 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 C18. Hereafter, this standard is referred to as *Standard C* in this guide.
 - C89—also known as C94, C90, and ANSI C. This standard is required when MISRA C is enabled in the compiler.
- Standard C++—also known as C++14. A well-established object-oriented programming language with a full-featured library well suited for modular programming. The IAR implementation of Standard C++ does not support exceptions and runtime type information (RTTI).

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. Both the strict and the relaxed mode might contain support for features in future versions of the C/C++ standards.

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

For more information about 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 RL78*.

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 RL78 DEVICES

The IAR C/C++ Compiler for RL78 supports all devices based on the standard Renesas RL78 microcontroller cores.

Note: In version 1.x of the compiler, the cores S1, S2, and S3 had the names RL78 0, RL78 1, and RL78 2.

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`.

Linker configuration files

The `r178\config` directory contains ready-made linker configuration files for all supported devices. The files have the filename extension `icf` 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 91, and for reference information, the chapter *The linker configuration file*.

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 `r178\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 RL78*.

EXAMPLES FOR GETTING STARTED

Example applications are provided with IAR Embedded Workbench. You can use these examples to get started using the development tools from IAR Systems. You can also use the examples as a starting point for your application project.

You can find the examples in the `r178\examples` directory. The examples are ready to be used as is. They are supplied with ready-made workspace files, together with source code files and all other related files. For information about how to run an example project, see the *IDE Project Management and Building Guide for RL78*.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the RL78 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 267 for additional information.

For more information, see the chapter *Extended keywords*. See also *Data storage and Functions*.

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 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 159.

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 assist you, compiler options, extended keywords, pragma directives, etc., are included.

CPU FEATURES AND CONSTRAINTS

The RL78 microcontroller is based on a 16-bit CISC Harvard architecture and has a full 16-bit data bus for RAM, CPU registers and peripheral units, and a 3-stage pipeline. It is designed for low-power applications and allows serial communication and ADC operation also in standby mode. Even though the RL78 microcontroller uses a CISC instruction set, the major part of the instructions are executed in 1–2 clock cycles.

The RL78 microcontroller has hardware facilities to assist with mathematics functions such as multiply, multiply/accumulate and divide. The multiply and MACC functions are executed in 1 and 2 clock cycles, respectively. The RL78 barrel shift function can execute a 16-bit `SHIFT` or `ROTATE` instruction in 1 clock cycle as well.

The compiler supports these hardware facilities by means of compiler options, extended keywords, pragma directives, etc.

MAPPING OF MEMORY

Embedded systems typically contain various types of memory, such as on-chip RAM, external DRAM or SRAM, 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 seldom accessed but must maintain its value after power off, so it 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 222.

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 91.

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 47. For an example, see *Accessing special function registers*, page 234.

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 79.

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, EPROM, or flash.

A C/C++ application further needs to initialize all global variables. This initialization is handled by the linker in conjunction with the system startup code. For more information, see *Application execution—an overview*, page 56.

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 into tasks that are 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.

See also *Managing a multithreaded environment*, page 157.

INTEROPERABILITY WITH OTHER BUILD TOOLS

You can use object files generated from tools from other vendors, provided that they follow the RL78 ABI specification, that you use the V2 calling convention, and that you do not use C++. In addition, the following limitations are known for modules using the Renesas standard C library, created by the Renesas RL78 compiler:

- Variadic functions like the `printf/scanf` family of standard C functions are not supported
- `strtok` is not supported
- You must use the Far code model and the Near data model in the IAR compiler.

The RL78 ABI specifies full compatibility for C object code, and for the C library.

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.

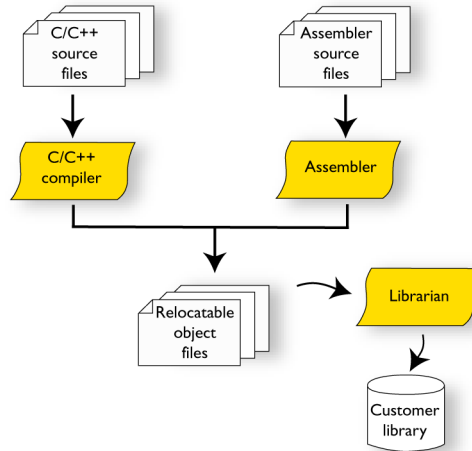
To become familiar with the process in practice, you should go through the tutorials available from the IAR Information Center.

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 industry-standard format ELF, including the DWARF format for debug information.

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 can then be assembled into object code. For more information about the IAR Assembler, see the *IAR Assembler User Guide for RL78*.

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 utility `iarchive`.

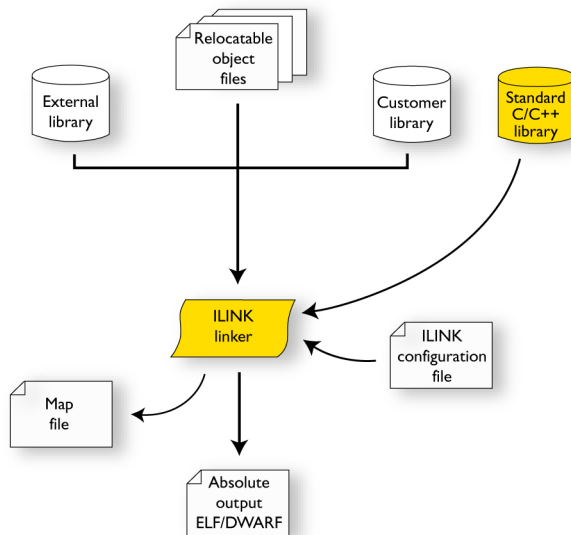
THE LINKING PROCESS

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 ILINK Linker (`ilinkr178.exe`) is used for building the final application. Normally, the linker requires the following information as input:

- Several object files and possibly certain libraries
- 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

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.

While 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.

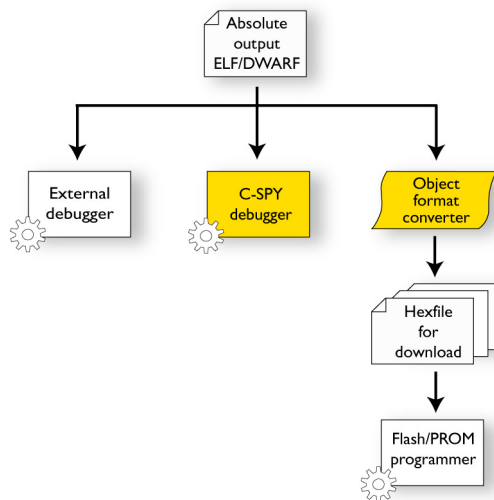
For more information about the procedure performed by the linker, see *The linking process in detail*, page 89.

AFTER LINKING

The IAR ILINK Linker produces an absolute object file in ELF format that contains the executable image. 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 ELF and DWARF.
- Programming to a flash/PROM using a flash/PROM programmer. Before this is possible, the actual bytes in the image must be converted into the standard Motorola 32-bit S-record format or the Intel Hex-32 format. For this, use `ielftool`, see *The IAR ELF Tool—ielftool*, page 460.

This illustration shows the possible uses of the absolute output ELF/DWARF file:



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

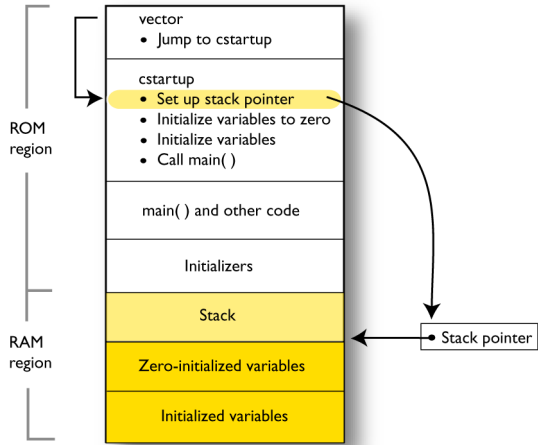
Initialization is executed when an application is started (the CPU is reset) but before the `main` function is entered. For simplicity, the initialization phase can be divided into:

- Hardware initialization, which as a minimum, generally initializes the stack pointer. The hardware initialization is typically performed in the system startup code `cstartup.s` and if required, by an extra low-level routine that you provide. It might include resetting/restarting 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.

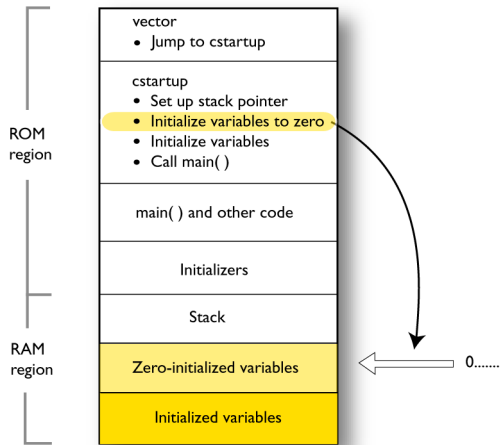
For a ROM/flash-based system, constants and functions are already placed in ROM. The linker has already divided the available RAM into different areas for variables, stack, heap, etc. All symbols placed in RAM must be initialized before the `main` function is called.

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:

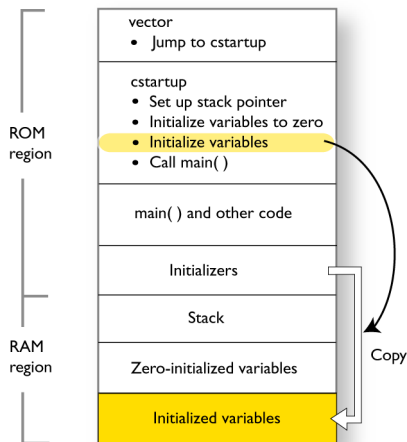


- Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:



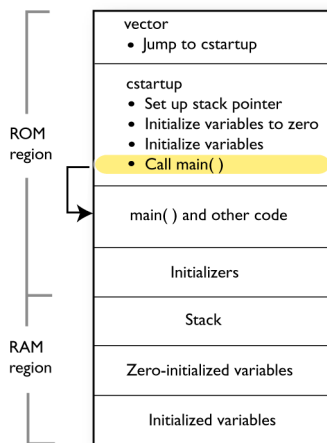
Typically, this is data referred to as *zero-initialized data*—variables declared as, for example, `int i = 0;`

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



Then, dynamically initialized static objects are constructed, such as C++ objects.

4 Finally, the `main` function is called:



For more information about each stage, see *System startup and termination*, page 141.

For more information about data initialization, see *Initialization at system startup*, page 94.

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 (RTOS). 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`, `quick_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 143.

Building applications—an overview

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

```
iccrl78 myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 61.

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

```
ilinkrl78 myfile.o myfile2.o -o a.out --config my_configfile.icf
```

In this example, `myfile.o` and `myfile2.o` are object files, and `my_configfile.icf` is the linker configuration file. The option `-o` specifies the name of the output file.

Note: By default, the label where the application starts is `__iar_program_start`. You can use the `--entry` 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, right-click in the **Build** messages window, and select **All** on the context menu.

Basic project configuration

This section gives an overview of the basic settings needed to generate the best code for the RL78 device you are using. You can specify the options either from the command line interface or in the IDE. On the command line, you must specify each option separately, but if you use the IDE, many options will be set automatically, based on your settings of some of the fundamental options.

You need to make settings for:

- Core
- Data model
- Code model

- Size of `double` floating-point type
- Optimization settings
- Runtime environment, see *Setting up the runtime environment*, page 125
- Customizing the ILINK configuration, see the chapter *Linking your application*.

In addition to these settings, you can use many other options and settings to fine-tune the result even further. For information about how to set options and for a list of all available options, see the chapters *Compiler options*, *Linker options*, and the *IDE Project Management and Building Guide for RL78*, respectively.

PROCESSOR CORE

The compiler supports these processor cores of the RL78 family:

Core	Description
S1	An S2 core with only one register bank and a multiplexed 8-bit bus.
S2	Does not have instructions to support a hardware multiplier/divider. Using a hardware multiplier/divider is instead supported by I/O register access.
S3 (default)	Has instructions to support a hardware multiplier/divider.

Table 3: Processor cores

Use the `--core` option to specify the processor core for your project; see `--core`, page 258.

Note: By default, the compiler uses bank 0 and never switches banks automatically. To make the compiler use banks 1–3, you must use the `#pragma bank` directive; see *bank*, page 355.

DATA MODEL

The compiler uses three data models, to make it easier to write code for applications with different data requirements:

- The *Near* data model can access data in the highest 64 Kbytes of data memory
- The *Far* data model can address data in the entire 1 Mbyte of data memory. It uses a faster but less flexible method than the Huge data model.
- The *Huge* data model can address data in the entire 1 Mbyte of data memory. It uses a slower but more flexible method than the Far data model.

For more information about the data models, see the chapter *Data storage*.

CODE MODEL

The compiler uses two code models that you can set on file- or function-level to control which function calls are generated by default:

- The *Near* code model has an upper limit of 64 Kbytes
- The *Far* code model can access the entire 1 Mbyte memory.

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, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

You can 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*.

Data storage

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

Introduction

The compiler supports RL78 devices with 1 Mbyte of continuous memory, ranging from 0x00000 to 0xFFFFF. Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

The compiler can access memory in different ways. The access methods range from generic but expensive methods that can access the full memory space, to a cheap method that can access the short address memory area. For more information about this, see *Memory types*, page 66.

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 73.

- *Global variables, module-static variables, and local variables declared static*

In this case, the memory is allocated once and for all. The word *static* in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Data models*, page 72 and *Memory types*, page 66.

- *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: 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 75.

Memory types

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

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 near addressing is called near 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 various memory types.

saddr

The short addressable area—saddr memory—consists of 256 bytes of data memory in the address range 0xFFE20–0xFFF1F.

A saddr object can only be placed in saddr memory. By using objects of this type, the code generated by the compiler to access them is minimized. This means a smaller footprint for the application, and faster execution at runtime.

See also *Short address working area*, page 74.

near

The near memory consists of 64 Kbytes of data memory. In hexadecimal notation, this is the address range 0xF0000–0xFFFFF.

A near object can only be placed in near memory.

far

The far memory consists of the whole 1 Mbyte of data memory. In hexadecimal notation, this is the address range 0x00000–0xFFFFF.

A far object can be placed anywhere in data memory, but the size of such an object is limited to 64 Kbytes.

huge

The huge memory consists of the whole 1 Mbyte of data memory. In hexadecimal notation, this is the address range 0x00000–0xFFFFF.

A huge object can be placed anywhere in data memory, and the size of such an object is not limited.

sfr

The special function register (SFR) area is located in the address range 0xFFFF0–0xFFFFF.

Special function registers are defined in the device-specific header files.

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
Saddr	<code>__saddr</code>	0xFFE20–0xFFF1F	2 bytes
Near (default)	<code>__near</code>	0xF0000–0xFFFFF	2 bytes
Far	<code>__far</code>	0x00000–0xFFFFF	3 bytes
Huge	<code>__huge</code>	0x00000–0xFFFFF	3 bytes

Table 4: Memory types and their corresponding keywords

Memory type	Keyword	Address range	Pointer size
SFR	<code>__sfr</code>	0xFFFF00-0xFFFFF	n/a

Table 4: Memory types and their corresponding keywords (Continued)

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 267 for additional information.

For more information about each keyword, see *Descriptions of extended keywords*, page 340.

Syntax for type attributes used on data objects

If you select the *uniform attribute syntax*, data type attributes use the same syntax rules as the type qualifiers `const` and `volatile`.

If not, data type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__near int i;
int __near j;
```

Both `i` and `j` are placed in near 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 __near * p;          /* pointer to integer in near memory */
int * __near p;         /* pointer in near memory */
__near int * p;        /* pointer in near memory */
```

The third case is interpreted differently when uniform attribute syntax is selected. If so, it is equivalent to the first case, just as would be the case if `const` or `volatile` were used correspondingly.

In all cases, if a memory attribute is not specified, an appropriate default memory type is used, which depends on the data model in use.

Using a type definition can sometimes make the code clearer:

```
typedef __near int d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in near 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=__near
int * q2;
```

The variable `q2` is placed in near memory.

For more examples of using memory attributes, see *More examples*, page 70.

For more information about the uniform attribute syntax, see *--uniform_attribute_syntax*, page 287 and *--no_uniform_attribute_syntax*, page 278.

Type definitions

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

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

/* Defines directly */
__far char aByte;
char __far *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 near memory is declared by:

```
int __near * 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 near memory. Like `MyPtr`, `MyPtr2` points to a character in far memory.

```
char __far * __near 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 RL78, the size of the near and far/huge pointers are 16 and 24 bits, respectively.

For more information about pointers, see *Pointer types*, page 328.

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 near memory.

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

__near struct MyStruct gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
    int mAlpha;
    __near 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 near memory is declared. The function returns a pointer to an integer in

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

<code>int MyA;</code>	A variable defined in default memory determined by the data model in use.
<code>int __near MyB;</code>	A variable in near memory.
<code>__far int MyC;</code>	A variable in far memory.
<code>int * MyD;</code>	A pointer stored in default memory. The pointer points to an integer in default memory.
<code>int __near * MyE;</code>	A pointer stored in default memory. The pointer points to an integer in near memory.
<code>int __near * __far MyF;</code>	A pointer stored in far memory pointing to an integer stored in near memory.
<code>int __far * MyFunction(int __near *);</code>	A declaration of a function that takes a parameter which is a pointer to an integer stored in near memory. The function returns a pointer to an integer stored in far 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 192.

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

All member functions except for constructors and destructors can be placed individually into a code memory in the same way as free functions.

For more information about C++ classes, see *Using IAR attributes with classes*, page 192.

Data models

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

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

The data model only specifies the default memory type. It is possible to override this for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 67.

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

SPECIFYING A 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 67.

This table summarizes the different data models:

Data model name	Default memory and pointer attribute	Placement of data
Near (default)	<code>__near</code>	The highest 64 Kbytes
Far	<code>__far</code>	The entire 1 Mbyte of memory
Huge	<code>__huge</code>	The entire 1 Mbyte of memory

Table 5: Data model characteristics



See the *IDE Project Management and Building Guide for RL78* 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 260.

The Near data model

The Near data model places data in the highest 64 Kbytes of memory. This memory can be accessed using 2-byte pointers, which means that only 16 bits are needed for pointer

storage. The default pointer type passed as a parameter will use one 16-bit register or 2 bytes on the stack.

The Far data model

The Far data model places data in the first 1 Mbyte of memory. This is the only memory that can be accessed using 3-byte pointers. The default pointer type passed as a parameter will use 4 bytes on the stack. By default, using this data model is faster but less flexible than the Huge data model.

The Huge data model

The Huge data model places data in the first 1 Mbyte of memory. This is the only memory that can be accessed using 3-byte pointers. The default pointer type passed as a parameter will use 4 bytes on the stack. By default, using this data model is slower but more flexible than the Far data 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, while 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).
- Canaries, used in stack-protected functions. See *Stack protection*, page 85.

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 206 and *Setting up stack memory*, page 109.

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 space. 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.

SHORT ADDRESS WORKING AREA

You can reserve space in saddr memory for a register variable work area, using the `--workseg_area` option, see *--workseg_area*, page 290. The most frequently used parameters and auto variables are stored in this area, up to the maximum size specified.

Avoid using the workseg area for functions that are called between a `setjmp` function call and its corresponding `longjmp` call, as the workseg area will likely become corrupt.

There is an overhead involved in saving and restoring the work area at each function entry and exit. This overhead can be omitted if the function is declared with the `__no_save` function attribute.

The workseg area is placed in the `.wrkseg` linker section, see `.wrkseg`, page 447.

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
Near	<code>__near</code>	NEAR_HEAP	near
Far	<code>__far</code>	FAR_HEAP	far
Huge	<code>__huge</code>	HUGE_HEAP	huge

Table 6: Heaps supported in memory types

To use a specific heap, add the memory attribute in front of `malloc`, `free`, `alloc`, and `realloc`, for example `__near_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 110.

POTENTIAL PROBLEMS

Applications that use heap-allocated data objects must be carefully designed, as 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 and memory attributes for function storage
- Primitives for interrupts, concurrency, and OS-related programming
- Inlining functions
- Stack protection

Function-related extensions

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

- Control the storage of functions in memory
- Use primitives for interrupts, concurrency, and OS-related programming
- 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 219. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models and memory attributes for function storage

Use *code models* to control how the compiler generates code for an application.

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. All code models produce code that is ROMable. The code model also specifies a default memory range for storing the functions.

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

These code models are available:

Near (default)	Function calls reach the first 64 Kbytes of memory
Far	Function calls reach the entire 1 Mbyte memory

Table 7: Code models

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



See the online help system 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 257.

USING FUNCTION MEMORY ATTRIBUTES

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

Function memory attribute	Address range	Pointer size	Description
<code>__callt</code>	0-0x0FFFF	2 bytes	The function can be called from anywhere in memory, using a vectored <code>CALLT</code> instruction.
<code>__near_func</code>	0-0x0FFFF	2 bytes	The function can be called from anywhere in memory. This is the default attribute in the Near code model.
<code>__far_func</code>	0-0xFFFFF	3 bytes	The function can be called from anywhere in memory. This is the default attribute in the Far code model.

Table 8: Function memory attributes

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For information about the restrictions, see *Casting*, page 329.

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

Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for RL78 provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords: `__interrupt`, `__monitor`
- The pragma directives: `#pragma vector`, `#pragma bank`
- The intrinsic functions: `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, `__set_interrupt_state`.

INTERRUPT FUNCTIONS

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

Interrupt service routines

In general, when an interrupt occurs in the code, the 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 RL78 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 RL78 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

For the RL78 microcontroller, the interrupt vector table always starts at the address `0x0` and is placed in the `.intvec` section. The interrupt vector is the offset into the interrupt vector table.

By default, the vector table is populated with a *default interrupt handler* which calls the `abort` function. For each interrupt source that has no explicit interrupt service routine, the default interrupt handler will be called. If you write your own service routine for a specific vector, that routine will override the default interrupt handler.

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

Defining an interrupt function—an example

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

```
#pragma vector = INTP2_vect /* Symbol defined in I/O header file
*/
__interrupt void MyInterruptRoutine(void)
{
    /* Do something */
}
```

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

Interrupt and C++ member functions

Only `static` member functions can be interrupt functions.

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 343.



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

Example of implementing a semaphore in C

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


```

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

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

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

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

        return 0;
    }
}

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

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

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

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

```

```

/* An example of using the semaphore. */

void MyProgram(void)
{
    GetLock();

    /* Do something here. */

    ReleaseLock();
}

```

Example of implementing a semaphore in C++

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

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

```

#include <intrinsics.h>

// Class for controlling critical blocks.

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

        // Disable all interrupts.
        __disable_interrupt();
    }

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

private:
    __istate_t mState;
};

```

```

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

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

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

private:
    static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
    static long nextStop = 100;

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

```

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 366.

- `--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 226.

For more information about function inlining optimization, see *Function inlining*, page 229.

Stack protection

In software, a stack buffer overflow occurs when a program writes to a memory address on the program's call stack outside of the intended data structure, which is usually a fixed-length buffer. The result is, almost always, corruption of nearby data, and it can even change which function to return to. If it is deliberate, it is often called stack smashing. One method to guard against stack buffer overflow is to use stack canaries, named for their analogy to the use of canaries in coal mines.

STACK PROTECTION IN THE IAR C/C++ COMPILER

The IAR C/C++ Compiler for RL78 supports stack protection.



To enable stack protection for functions considered needing it, use the compiler option `--stack_protection`. For more information, see *--stack_protection*, page 285.

The IAR Systems implementation of stack protection uses a heuristic to determine whether a function needs stack protection or not. If any defined local variable has the array type or a structure type that contains a member of array type, the function will need stack protection. In addition, if the address of any local variable is propagated outside of a function, such a function will also need stack protection.

If a function needs stack protection, the local variables are sorted to let the variables with array type to be placed as high as possible in the function stack block. After those

variables, a canary element is placed. The canary is initialized at function entrance. The initialization value is taken from the global variable `__stack_chk_guard`. At function exit, the code verifies that the canary element still contains the original value. If not, the function `__stack_chk_fail` is called.

USING STACK PROTECTION IN YOUR APPLICATION

To use stack protection, you must define these objects in your application:

- `extern uint16_t __stack_chk_guard`

The global variable `__stack_chk_guard` must be initialized prior to first use. If the initialization value is randomized, it will be more secure.

- `__nounwind __noreturn void __stack_chk_fail(void)`

The purpose of the function `__stack_chk_fail` is to notify about the problem and then terminate the application.

Note: The return address from this function will point into the function that failed.

The file `stack_protection.c` in the directory `rl78\src\lib\runtime` can be used as a template for both `__stack_chk_guard` and `__stack_chk_fail`.

Linking using ILINK

- Linking—an overview
- Modules and sections
- 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 ILINK 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 selected parts of one or more object libraries to produce an executable image in the industry-standard format *Executable and Linking Format* (ELF).

The linker will automatically load only those library modules—user libraries and Standard C or C++ library variants—that are actually needed by the application you are linking. Furthermore, the linker eliminates duplicate sections and sections that are not required.

The linker uses a *configuration file* where you can specify separate locations for code and data areas of your target system memory map. This file also supports automatic handling of the application's initialization phase, which means initializing global variable areas and code areas by copying initializers and possibly decompressing them as well.

The final output produced by ILINK is an absolute object file containing the executable image in the ELF (including DWARF for debug information) format. The file can be downloaded to C-SPY or any other compatible debugger that supports ELF/DWARF, or it can be stored in EPROM or flash.

To handle ELF files, various tools are included. For information about included utilities, see *Specific ELF tools*, page 46.

Modules and sections

Each relocatable object file contains one module, which consists of:

- Several sections of code or data
- Runtime attributes specifying various types of information, for example, the version of the runtime environment
- Optionally, debug information in DWARF format
- A symbol table of all global symbols and all external symbols used.

A *section* is a logical entity containing a piece of data or code that should be placed at a physical location in memory. A section can consist of several *section fragments*, typically one for each variable or function (symbols). A section can be placed either in RAM or in ROM. In a normal embedded application, sections that are placed in RAM do not have any content, they only occupy space.

Each section has a name and a type attribute that determines the content. The type attribute is used (together with the name) for selecting sections for the ILINK configuration.

The main purpose of section attributes is to distinguish between sections that can be placed in ROM and sections that must be placed in RAM:

<code>ro readonly</code>	ROM sections
<code>rw readwrite</code>	RAM sections

In each category, sections can be further divided into those that contain code and those that contain data, resulting in four main categories:

<code>ro code</code>	Normal code
<code>ro data</code>	Constants
<code>rw code</code>	Code copied to RAM
<code>rw data</code>	Variables

`readwrite data` also has a subcategory—`zi|zeroinit`—for sections that are zero-initialized at application startup.

Note: In addition to these section types—sections that contain the code and data that are part of your application—a final object file will contain many other types of sections,

for example, sections that contain debugging information or other type of meta information.

A section is the smallest linkable unit—but if possible, ILINK can exclude smaller units—section fragments—from the final application. For more information, see *Keeping modules*, page 109, and *Keeping symbols and sections*, page 109.

At compile time, data and functions are placed in different sections. At link time, one of the most important functions of the linker is to assign addresses to the various sections used by the application.

The IAR build tools have many predefined section names. For more information about each section, see the chapter *Section reference*.

You can group sections together for placement by using blocks. See *define block directive*, page 415.

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 become an executable application, they must be *linked*.

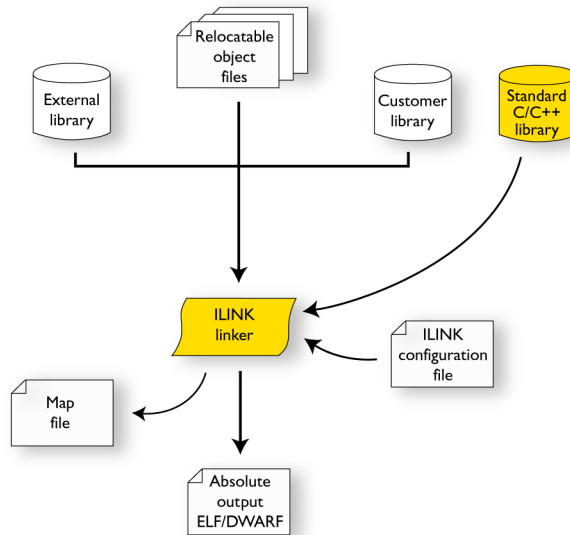
The linker is used for the link process. It normally performs the following procedure (note that some of the steps can be turned off by command line options or by directives in the linker configuration file):

- Determine which modules to include in the application. Modules provided in object files are always included. A module in a library file is only included if it provides a definition for a global symbol that is referenced from an included module.
- Select which standard library files to use. The selection is based on attributes of the included modules. These libraries are then used for satisfying any still outstanding undefined symbols.
- Handle symbols with more than one definition. If there is more than one non-weak definition, an error is emitted. Otherwise, one of the definitions is picked (the non-weak one, if there is one) and the others are suppressed. Weak definitions are typically used for inline and template functions. If you need to override some of the non-weak definitions from a library module, you must ensure that the library module is not included (typically by providing alternate definitions for all the symbols your application uses in that library module).
- Determine which sections/section fragments from the included modules to include in the application. Only those sections/section fragments that are actually needed by the application are included. There are several ways to determine of which sections/section fragments that are needed, for example, the `__root` object

attribute, the `#pragma required` directive, and the `keep` linker directive. In case of duplicate sections, only one is included.

- Where appropriate, arrange for the initialization of initialized variables and code in RAM. The `initialize` directive causes the linker to create extra sections to enable copying from ROM to RAM. Each section that will be initialized by copying is divided into two sections—one for the ROM part, and one for the RAM part. If manual initialization is not used, the linker also arranges for the startup code to perform the initialization.
- Determine where to place each section according to the section placement directives in the *linker configuration file*. Sections that are to be initialized by copying appear twice in the matching against placement directives, once for the ROM part and once for the RAM part, with different attributes.
- Produce an absolute file that contains the executable image and any debug information provided. The contents of each needed section in the relocatable input files is calculated using the relocation information supplied in its file and the addresses determined when placing sections. This process can result in one or more relocation failures if some of the requirements for a particular section 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, produce a map file that lists the result of the section placement, the address of each global symbol, and finally, a summary of memory usage for each module and library.

This illustration shows the linking process:



During the linking, ILINK might produce error and logging messages on `stdout` and `stderr`. The log messages are useful for understanding why an application was linked as it was. For example, why a module or section (or section fragment) was included.

Note: To see the actual content of an ELF object file, use `ielfdumpr178`. See *The IAR ELF Dumper—ielfdump*, page 462.

Placing code and data—the linker configuration file

The placement of sections in memory is performed by the IAR ILINK Linker. It uses the *linker configuration file* where you can define how ILINK should treat each section and how they should be placed into the available memories.

A typical linker configuration file contains definitions of:

- Available addressable memories
- Populated regions of those memories
- How to treat input sections
- Created sections
- How to place sections into the available regions.

The file consists of a sequence of declarative directives. This means that the linking process will be governed by all directives at the same time.

To use the same source code with different derivatives, just rebuild the code with the appropriate configuration file.

A SIMPLE EXAMPLE OF A CONFIGURATION FILE

Assume a simple 32-bit architecture that has these memory prerequisites:

- There are 4 Gbytes of addressable memory.
- There is ROM memory in the address range 0x0000–0x10000.
- There is RAM memory in the range 0x20000–0x30000.
- The stack has an alignment of 8.
- The system startup code must be located at a fixed address.

A simple configuration file for this assumed architecture can look like this:

```
/* The memory space denoting the maximum possible amount
   of addressable memory */
define memory Mem with size = 4G;

/* Memory regions in an address space */
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* Create a stack */
define block STACK with size = 0x1000, alignment = 2 { };

/* Handle initialization */
initialize by copy { readwrite }; /* Initialize RW sections */

/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };

/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                             ROM: .rodata and .data_init */
place in RAM { readwrite, /* Place .data, .bss, and .noinit */
               block STACK }; /* and STACK */
```

This configuration file defines one addressable memory *Mem* with the maximum of 4 Gbytes of memory. Furthermore, it defines a ROM region and a RAM region in *Mem*, namely ROM and RAM. Each region has the size of 64 Kbytes.

The file then creates an empty block called *STACK* with a size of 4 Kbytes in which the application stack will reside. To create a *block* is the basic method which you can use to

get detailed control of placement, size, etc. It can be used for grouping sections, but also as in this example, to specify the size and placement of an area of memory.

Next, the file defines how to handle the initialization of variables, read/write type (`readwrite`) sections. In this example, the initializers are placed in ROM and copied at startup of the application to the RAM area. By default, ILINK may compress the initializers if this appears to be advantageous.

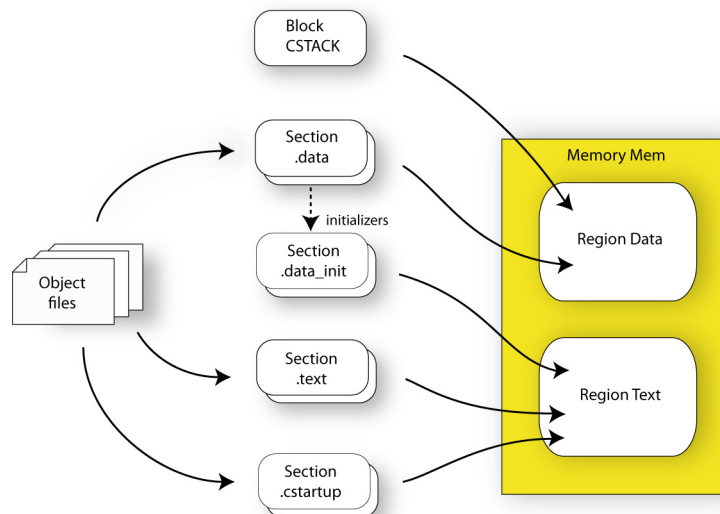
The last part of the configuration file handles the actual placement of all the sections into the available regions. First, the startup code—defined to reside in the read-only (`readonly`) section `.cstartup`—is placed at the start of the ROM region, that is at address `0x10000`.

Note: The part within `{ }` is referred to as *section selection* and it selects the sections for which the directive should be applied to. Then the rest of the read-only sections are placed in the ROM region.

Note: The section selection `{ readonly section .cstartup }` takes precedence over the more generic section selection `{ readonly }`.

Finally, the read/write (`readwrite`) sections and the `STACK` block are placed in the RAM region.

This illustration gives a schematic overview of how the application is placed in memory:



In addition to these standard directives, a configuration file can contain directives that define how to:

- Map a memory that can be addressed in multiple ways
- Handle conditional directives
- Create symbols with values that can be used in the application
- More in detail, select the sections a directive should be applied to
- More in detail, initialize code and data.

For more details and examples about customizing the linker configuration file, see the chapter *Linking your application*.

For more information about the linker configuration file, see the chapter *The linker configuration file*.

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. This value is either an explicit value assigned to the variable, or if no value is given, it is cleared to zero. In the compiler, there are exceptions to this rule, for example variables declared `__no_init`, which are not initialized at all.

The compiler generates a specific type of section for each type of variable initialization:

Categories of declared data	Source	Section type	Section name element*	Section content
Zero-initialized data	<code>int i;</code>	Read/write data, zero-init	<code>.bss</code>	None
Zero-initialized data	<code>int i = 0;</code>	Read/write data, zero-init	<code>.bss</code>	None
Initialized data (non-zero)	<code>int i = 6;</code>	Read/write data	<code>.data</code>	The initializer
Non-initialized data	<code>__no_init int i;</code>	Read/write data, zero-init	<code>.bss.noinit</code>	None
Constants	<code>const int i = 6;</code>	Read-only data	<code>.const</code>	The constant

Table 9: Sections holding initialized data

* The name of the section contains this element plus a letter that denotes the memory associated with the section: `f` for far, `h` for huge and `s` for saddr. This letter can come before or after the section name element described in this table. For near memory, there

is no such extra letter. For more information about possible section names, see *Summary of sections*, page 437.

For information about all supported sections, see the chapter *Section reference*.

THE INITIALIZATION PROCESS

Initialization of data is handled by ILINK and the system startup code in conjunction.

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

- Sections that should be zero-initialized, or not initialized at all (`__no_init`) are handled automatically by ILINK.
- Sections that should be initialized, except for zero-initialized sections, should be listed in an `initialize` directive.

Normally during linking, a section that should be initialized is split into two sections, where the original initialized section will keep the name. The contents are placed in the new initializer section, which will get the original name suffixed with `_init`. The initializers should be placed in ROM and the initialized sections in RAM, by means of placement directives. The most common example is the `.data` section which the linker splits into `.data` and `.data_init`.

- Sections that contains constants should not be initialized—they should only be placed in flash/ROM.

In the linker configuration file, it can look like this:

```
/* Handle initialization */
initialize by copy { readwrite }; /* Initialize RW sections */

/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };

/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                           ROM: .rodata and .data_init */
place in RAM { readwrite, /* Place .data, .bss, and .noinit */
              block STACK }; /* and STACK */
```

Note: When compressed initializers are used (see *initialize directive*, page 418), the contents sections (that is, sections with the `_init` suffix) are not listed as separate sections in the map file. Instead, they are combined into aggregates of “initializer bytes”. You can place the contents sections the usual way in the linker configuration file, however, this affects the placement—and possibly the number—of the “initializer bytes” aggregates.

For more information about and examples of how to configure the initialization, see *Linking considerations*, page 105.

C++ DYNAMIC INITIALIZATION

The compiler places subroutine pointers for performing C++ dynamic initialization into sections of the ELF section types `SHT_PREINIT_ARRAY` and `SHT_INIT_ARRAY`. By default, the linker will place these into a linker-created block, ensuring that all sections of the section type `SHT_PREINIT_ARRAY` are placed before those of the type `SHT_INIT_ARRAY`. If any such sections were included, code to call the routines will also be included.

The linker-created blocks are only generated if the linker configuration does not contain section selector patterns for the `preinit_array` and `init_array` section types. The effect of the linker-created blocks will be very similar to what happens if the linker configuration file contains this:

```
define block SHT$$PREINIT_ARRAY { preinit_array };
define block SHT$$INIT_ARRAY { init_array };
define block CPP_INIT with fixed order { block
                                     SHT$$PREINIT_ARRAY,
                                     block SHT$$INIT_ARRAY };
```

If you put this into your linker configuration file, you must also mention the `CPP_INIT` block in one of the section placement directives. If you wish to select where the linker-created block is placed, you can use a section selector with the name `".init_array"`.

See also *section-selectors*, page 427.

ADDITIONAL COMPILER-GENERATED SECTIONS

The compiler uses a set of internally generated sections, which are used for storing information that is vital to the operation of your application.

- The `.switch` and `.switchf` sections which contain data statements used in the `switch` library routines. These tables are encoded in such a way as to use as little space as possible.
- The `.wrkseg` section which holds space reserved by the `--workseg_area` option and space used by floating-point operations.

In the linker configuration file, it can look like this:

```
"ROMPAGE":place in ROM_page      { ro section .const,
                                   ro section .switch };
```


Stack usage analysis

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

In the `r178\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 `--enable_stack_usage`, page 300.

- 2 Enable the linker map file:



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



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

- 3 Link your project.

Note: The linker will issue warnings related to stack usage under certain circumstances, see *Situations where warnings are issued*, page 102.

- 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 98.
- 5 For more details, analyze the call graph log, see *Call graph log*, page 102.

Note: There are limitations and sources of inaccuracy in the analysis, see *Limitations*, page 101.

You might need to specify more information to the linker to get a more representative result. See *Specifying additional stack usage information*, page 100



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 `--stack_usage_control`, page 315.

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

```
check that size(block 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` block:

- 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 431 and *Stack considerations*, page 206.

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
***
```

Call Graph Root Category	Max Use	Total Use
-----	-----	-----
interrupt	104	136
Program entry	168	168
Program entry		
"__iar_program_start": 0x000085ac		
Maximum call chain		168 bytes
"__iar_program_start"		0
"__cmain"		0
"main"		8
"printf"		24
"_PrintfTiny"		56
"_Prout"		16
"putchar"		16
"__write"		0
"__dwrite"		0
"__iar_sh_stdout"		24
"__iar_get_ttio"		24
"__iar_lookup_ttioh"		0
interrupt		
"FaultHandler": 0x00008434		
Maximum call chain		32 bytes
"FaultHandler"		32
interrupt		
"IRQHandler": 0x00008424		
Maximum call chain		104 bytes
"IRQHandler"		24
"do_something" in suexample.o [1]		80

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 450.

- 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 450.

- 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 452 and *calls*, page 357.

- 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 450 and *call_graph_root*, page 358.

- 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.o] 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 RL78*.

For more information, see the chapter *The stack usage control file*.

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 or produced by non-IAR tools. 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 RL78*.
- 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.
- 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: 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. In particular, the set of possible destinations for a virtual function call in C++ might sometimes include implementations of the function in question which cannot, in fact, be called from that point in the code.



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 call_graph`).

Example output:

```

Program entry:
0 __iar_program_start [168]
  0 __cmain [168]
    0 __iar_data_init3 [16]
      8 __iar_zero_init3 [8]
        16 - [0]
      8 __iar_copy_init3 [8]
        16 - [0]
    0 __low_level_init [0]
  0 main [168]
    8 printf [160]
      32 _PrintfTiny [136]
        88 _Prout [80]
          104 putchar [64]
            120 __write [48]
              120 __dwrite [48]
                120 __iar_sh_stdout [48]
                  144 __iar_get_ttio [24]
                    168 __iar_lookup_ttioh [0]
                      120 __iar_sh_write [24]
                        144 - [0]
              88 __aeabi_uidiv [0]
                88 __aeabi_idiv0 [0]
            88 strlen [0]
          0 exit [8]
            0 _exit [8]
              0 __iar_close_ttio [8]
                8 __iar_lookup_ttioh [0] ***
            0 _exit [8] ***

```

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.

CALL GRAPH XML OUTPUT

The linker can also produce a call graph file in XML format. This file contains one node for each function in your application, with the stack usage and call information relevant

to that function. It is intended to be input for post-processing tools and is not particularly human-readable.

For more information about the XML format used, see the `callGraph.txt` file in your product installation.

Linking your application

- Linking considerations
- Hints for troubleshooting
- Checking module consistency
- Linker optimizations

Linking considerations

Before you can link your application, you must set up the configuration required by ILINK. Typically, you must consider:

- *Choosing a linker configuration file*, page 105
- *Defining your own memory areas*, page 106
- *Placing sections*, page 107
- *Reserving space in RAM*, page 108
- *Keeping modules*, page 109
- *Keeping symbols and sections*, page 109
- *Application startup*, page 109
- *Setting up stack memory*, page 109
- *Setting up heap memory*, page 110
- *Setting up the atexit limit*, page 110
- *Changing the default initialization*, page 110
- *Interaction between ILINK and the application*, page 114
- *Standard library handling*, page 115
- *Producing other output formats than ELF/DWARF*, page 115
- *Linking object files produced by the Renesas CC-RL compiler*, page 115

CHOOSING A LINKER CONFIGURATION FILE

The `config` directory contains ready-made linker configuration files for all supported devices. The files contain the information required by ILINK. The only change, if any, you will normally have to make to the supplied configuration file is to customize the start and end addresses of each region so they fit 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.

Each project in the IDE should have a reference to one, and only one, linker configuration file. This file can be edited, but for the majority of all projects it is sufficient to configure the vital parameters in **Project>Options>Linker>Config**.

DEFINING YOUR OWN MEMORY AREAS

The default configuration file that you selected has predefined ROM and RAM regions. This example will be used as a starting-point for all further examples in this chapter:

```
/* Define the addressable memory */
define memory Mem with size = 4G;

/* Define a region named ROM with start address 0 and to be 64
Kbytes large */
define region ROM = Mem:[from 0 size 0x10000];

/* Define a region named RAM with start address 0x20000 and to be
64 Kbytes large */
define region RAM = Mem:[from 0x20000 size 0x10000];
```

Each region definition must be tailored for the actual hardware.

To find out how much of each memory that was filled with code and data after linking, inspect the memory summary in the map file (command line option `--map`).

Adding an additional region

To add an additional region, use the `define region` directive, for example:

```
/* Define a 2nd ROM region to start at address 0x80000 and to be
128 Kbytes large */
define region ROM2 = Mem:[from 0x80000 size 0x20000];
```

Merging different areas into one region

If the region is comprised of several areas, use a region expression to merge the different areas into one region, for example:

```
/* Define the 2nd ROM region to have two areas. The first with
the start address 0x80000 and 128 Kbytes large, and the 2nd with
the start address 0xC0000 and 32 Kbytes large */
define region ROM2 = Mem:[from 0x80000 size 0x20000]
                    | Mem:[from 0xC0000 size 0x08000];
```

or equivalently

```
define region ROM2 = Mem:[from 0x80000 to 0xC7FFF]
                    -Mem:[from 0xA0000 to 0xBFFFF];
```

PLACING SECTIONS

The default configuration file that you selected places all predefined sections in memory, but there are situations when you might want to modify this. For example, if you want to place the section that holds constant symbols in the `CONSTANT` region instead of in the default place. In this case, use the `place in` directive, for example:

```
/* Place sections with readonly content in the ROM region */
place in ROM {readonly};

/* Place the constant symbols in the CONSTANT region */
place in CONSTANT {readonly section .rodata};
```

Note: Placing a section—used by the IAR build tools—in a different memory which use a different way of referring to its content, will fail.

For the result of each placement directive after linking, inspect the placement summary in the map file (the command line option `--map`).

Placing a section at a specific address in memory

To place a section at a specific address in memory, use the `place at` directive, for example:

```
/* Place section .vectors at address 0 */
place at address Mem:0x0 {readonly section .vectors};
```

Placing a section first or last in a region

To place a section first or last in a region is similar, for example:

```
/* Place section .vectors at start of ROM */
place at start of ROM {readonly section .vectors};
```

Declare and place your own sections

To declare new sections—in addition to the ones used by the IAR build tools—to hold specific parts of your code or data, use mechanisms in the compiler and assembler. For example:

```
/* Place a variable in that section. */
const short MyVariable @ "MYOWNSECTION" = 0xF0F0;
```

This is the corresponding example in assembler language:

```
name      createSection
section MYOWNSECTION:CONST ; Create a section,
                                ; and fill it with
dc16     0xF0F0              ; constant bytes.
end
```

To place your new section, the original `place in ROM {readonly}`; directive is sufficient.

However, to place the section `MyOwnSection` explicitly, update the linker configuration file with a `place in` directive, for example:

```
/* Place MyOwnSection in the ROM region */
place in ROM {readonly section MyOwnSection};
```

RESERVING SPACE IN RAM

Often, an application must have an empty uninitialized memory area to be used for temporary storage, for example, a heap or a stack. It is easiest to achieve this at link time. You must create a block with a specified size and then place it in a memory.

In the linker configuration file, it can look like this:

```
define block TempStorage with size = 0x1000, alignment = 4 { };
place in RAM { block TempStorage };
```

To retrieve the start of the allocated memory from the application, the source code could look like this:

```
/* Define a section for temporary storage. */
#pragma section = "TempStorage"
char *GetTempStorageStartAddress()
{
    /* Return start address of section TempStorage. */
    return __section_begin("TempStorage");
}
```

KEEPING MODULES

If a module is linked as an object file, it is always kept. That is, it will contribute to the linked application. However, if a module is part of a library, it is included only if it is symbolically referred to from other parts of the application. This is true, even if the library module contains a root symbol. To assure that such a library module is always included, use `iarchive` to extract the module from the library, see *The IAR Archive Tool—iarchive*, page 457.

For information about included and excluded modules, inspect the log file (the command line option `--log modules`).

For more information about modules, see *Modules and sections*, page 88.

KEEPING SYMBOLS AND SECTIONS

By default, ILINK removes any sections, section fragments, and global symbols that are not needed by the application. To retain a symbol that does not appear to be needed—or actually, the section fragment it is defined in—you can either use the root attribute on the symbol in your C/C++ or assembler source code, or use the ILINK option `--keep`. To retain sections based on attribute names or object names, use the directive `keep` in the linker configuration file.

To prevent ILINK from excluding sections and section fragments, use the command line options `--no_remove` or `--no_fragments`, respectively.

For information about included and excluded symbols and sections, inspect the log file (the command line option `--log sections`).

For more information about the linking procedure for keeping symbols and sections, see *The linking process*, page 55.

APPLICATION STARTUP

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

To change the start point of the application to another label, use the ILINK option `--entry`, see `--entry`, page 301.

SETTING UP STACK MEMORY

The size of the `CSTACK` block is defined in the linker configuration file. To change the allocated amount of memory, set the symbol `_STACK_SIZE` to the required value:

```
define block CSTACK with alignment = 2, size = _STACK_SIZE
    { rw section CSTACK };
```

Specify an appropriate size for your application.

For more information about the stack, see *Stack considerations*, page 206.

SETTING UP HEAP MEMORY

The size of the heap is defined in the linker configuration file as a block. To change the allocated amount of heap memory, set the symbols `_NEAR_HEAP_SIZE`, `_FAR_HEAP_SIZE`, and `_HUGE_HEAP_SIZE` to the required values:

```
define block NEAR_HEAP with alignment = 2,
    size = _NEAR_HEAP_SIZE { };
define block FAR_HEAP with alignment = 2,
    size = _FAR_HEAP_SIZE { };
define block HUGE_HEAP with alignment = 2,
    size = _HUGE_HEAP_SIZE { };
```

If you use heaps, you must allocate at least 50 bytes each for them.

SETTING UP THE ATEXTIT LIMIT

By default, the `atexit` function can be called a maximum of 32 times from your application. To either increase or decrease this number, add a line to your configuration file. For example, to reserve room for 10 calls instead, write:

```
define symbol __iar_maximum_atexit_calls = 10;
```

CHANGING THE DEFAULT INITIALIZATION

By default, memory initialization is performed during application startup. ILINK sets up the initialization process and chooses a suitable packing method. If the default initialization process does not suit your application and you want more precise control over the initialization process, these alternatives are available:

- Suppressing initialization
- Choosing the packing algorithm
- Manual initialization
- Initializing code—copying ROM to RAM.

For information about the performed initializations, inspect the log file (the command line option `--log initialization`).

Suppressing initialization

If you do not want the linker to arrange for initialization by copying, for some or all sections, make sure that those sections do not match a pattern in an `initialize by copy` directive—or use an `except` clause to exclude them from matching. If you do not

want any initialization by copying at all, you can omit the `initialize by copy` directive entirely.

This can be useful if your application, or just your variables, are loaded into RAM by some other mechanism before application startup.

Choosing a packing algorithm

To override the default packing algorithm, write for example:

```
initialize by copy with packing = lz77 { readwrite };
```

For more information about the available packing algorithms, see *initialize directive*, page 418.

Manual initialization

In the usual case, the `initialize by copy` directive is used for making the linker arrange for initialization by copying—with or without packing—of sections with content at application startup. The linker achieves this by logically creating an initialization section for each such section, holding the content of the section, and turning the original section into a section without content. Then, the linker adds table elements to the initialization table so that the initialization will be performed at application startup. You can use `initialize manually` to suppress the creation of table elements to take control over when and how the elements are copied. This is useful for overlays, but also in other circumstances.

For sections without content (zero-initialized sections), the situation is reversed. The linker arranges for zero initialization of all such sections at application startup, except for those that are mentioned in a `do not initialize` directive.

Simple copying example with an automatic block

Assume that you have some initialized variables in `MYSECTION`. If you add this directive to your linker configuration file:

```
initialize manually { section MYSECTION };
```

you can use this source code example to initialize the section:

```
#pragma section = "MYSECTION"
#pragma section = "MYSECTION_init"

void DoInit()
{
    char * from = __section_begin("MYSECTION_init");
    char * to   = __section_begin("MYSECTION");
    memcpy(to, from, __section_size("MYSECTION"));
}
```

This piece of source code takes advantage of the fact that if you use `__section_begin` (and related operators) with a section name, an automatic block is created by the linker for those sections.

Note: Automatic blocks override the normal section selection process and forces everything that matches the section name to form on block.

Example with explicit blocks

Assume that you instead of needing manual initialization for variables in a specific section, you need it for all initialized variables from a particular library. In that case, you must create explicit blocks for both the variables and the content. Like this:

```
initialize manually      { section .data      object mylib.a };
define block MYBLOCK    { section .data      object mylib.a };
define block MYBLOCK_init { section .data_init object mylib.a };
```

You must also place the two new blocks using one of the section placement directives, the block `MYBLOCK` in RAM and the block `MYBLOCK_init` in ROM.

Then you can initialize the sections using the same source code as in the previous example, only with `MYBLOCK` instead of `MYSECTION`.

Overlay example

This is a simple overlay example that takes advantage of automatic block creation:

```
initialize manually { section MYOVERLAY* };

define overlay MYOVERLAY { section MYOVERLAY1 };
define overlay MYOVERLAY { section MYOVERLAY2 };
```


You must also place `overlay MYOVERLAY` somewhere in RAM. The copying could look like this:

```
#pragma section = "MYOVERLAY"
#pragma section = "MYOVERLAY1_init"
#pragma section = "MYOVERLAY2_init"

void SwitchToOverlay1()
{
    char * from = __section_begin("MYOVERLAY1_init");
    char * to   = __section_begin("MYOVERLAY");
    memcpy(to, from, __section_size("MYOVERLAY1_init"));
}

void SwitchToOverlay2()
{
    char * from = __section_begin("MYOVERLAY2_init");
    char * to   = __section_begin("MYOVERLAY");
    memcpy(to, from, __section_size("MYOVERLAY2_init"));
}
```

Initializing code—copying ROM to RAM

Sometimes, an application copies pieces of code from flash/ROM to RAM. You can direct the linker to arrange for this to be done automatically at application startup, or do it yourself at some later time using the techniques described in *Manual initialization*, page 111.

You need to list the code sections that should be copied in an `initialize by copy` directive. The easiest way is usually to place the relevant functions in a particular section—for example, `RAMCODE`—and add `section RAMCODE` to your `initialize by copy` directive. For example:

```
initialize by copy { rw, section RAMCODE };
```

If you need to place the `RAMCODE` functions in some particular location, you must mention them in a placement directive, otherwise they will be placed together with other read/write sections.

If you need to control the manner and/or time of copying, you must use an `initialize manually` directive instead. See *Manual initialization*, page 111.

Running all code from RAM

If you want to copy the entire application from ROM to RAM at program startup, use the `initilize by copy` directive, for example:

```
initialize by copy { readonly, readwrite };
```

The `readwrite` pattern will match all statically initialized variables and arrange for them to be initialized at startup. The `readonly` pattern will do the same for all read-only code and data, except for code and data needed for the initialization.

Because the function `__low_level_init`, if present, is called before initialization, it and anything it needs, will not be copied from ROM to RAM either. In some circumstances—for example, if the ROM contents are no longer available to the program after startup—you might need to avoid using the same functions during startup and in the rest of the code.

If anything else should not be copied, include it in an `except` clause. This can apply to, for example, the interrupt vector table.

It is also recommended to exclude the C++ dynamic initialization table from being copied to RAM, as it is typically only read once and then never referenced again. For example, like this:

```
initialize by copy { readonly, readwrite }
    except { section .intvec,          /* Don't copy
                                         interrupt table */
            section .init_array }; /* Don't copy
                                         C++ init table */
```

INTERACTION BETWEEN ILINK AND THE APPLICATION

ILINK provides the command line options `--config_def` and `--define_symbol` to define symbols which can be used for controlling the 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. For more information, see *Interaction between the tools and your application*, page 207.

To change a reference to one symbol to another symbol, use the ILINK command line option `--redirect`. 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 example, how to choose the DLIB formatter for the standard library functions `printf` and `scanf`.

The compiler generates mangled names to represent complex C/C++ symbols. If you want to refer to these symbols from assembler source code, you must use the mangled names.

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

For more information, see *Interaction between the tools and your application*, page 207.

STANDARD LIBRARY HANDLING

By default, ILINK determines automatically which variant of the standard library to include during linking. The decision is based on the sum of the runtime attributes available in each object file and the library options passed to ILINK.

To disable the automatic inclusion of the library, use the option `--no_library_search`. In this case, you must explicitly specify every library file to be included. For information about available library files, see *Prebuilt runtime libraries*, page 132.

PRODUCING OTHER OUTPUT FORMATS THAN ELF/DWARF

ILINK can only produce an output file in the ELF/DWARF format. To convert that format into a format suitable for programming PROM/flash, see *The IAR ELF Tool—ielftool*, page 460.

LINKING OBJECT FILES PRODUCED BY THE RENESAS CC-RL COMPILER

ILINK does not fully support linking sections with the attribute `not-separated` in object files produced by the Renesas CC-RL compiler, like the section `text_unit64kp`. To prevent ILINK from separating such section parts, you can either:

- 1 In either C or assembler source code, take the size of a section or the start or end address of a section.
- 2 In the linker configuration file, put the section(s) in a separate block that you place in the correct memory using the `define block` and `place in` directives, like this:

```
define block MY_BLOCK { [ro|rw] section MY_SEG };
"ROMFAR":place in ROM_far { block MY_BLOCK };
```

Hints for troubleshooting

ILINK has several features that can help you manage code and data placement correctly, for example:

- Messages at link time, for examples when a relocation error occurs
- The `--log` option that makes ILINK log information to `stdout`, which can be useful to understand why an executable image became the way it is, see `--log`, page 305
- The `--map` option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see `--map`, page 307.

RELOCATION ERRORS

For each instruction that cannot be relocated correctly, ILINK will generate a *relocation error*. This can occur for instructions where the target is out of reach or is of an incompatible type, or for many other reasons.

A relocation error produced by ILINK can look like this:

```
Error[Lp002]: relocation failed: out of range or illegal value
Kind      : R_XXX_YYY[0x1]
Location  : 0x40000448
           "myfunc" + 0x2c
           Module:  somecode.o
           Section: 7 (.text)
           Offset:  0x2c
Destination: 0x9000000c
           "read"
           Module:  read.o(iolib.a)
           Section: 6 (.text)
           Offset:  0x0
```

The message entries are described in this table:

Message entry	Description
Kind	The relocation directive that failed. The directive depends on the instruction used.
Location	The location where the problem occurred, described with the following details: <ul style="list-style-type: none"> • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, <code>0x40000448</code> and <code>"myfunc" + 0x2c</code>. • The module, and the file. In this example, the module <code>somecode.o</code>. • The section number and section name. In this example, section number <code>7</code> with the name <code>.text</code>. • The offset, specified in number of bytes, in the section. In this example, <code>0x2c</code>.

Table 10: Description of a relocation error

Message entry	Description
Destination	<p>The target of the instruction, described with the following details:</p> <ul style="list-style-type: none"> • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, <code>0x9000000c</code> and "read"—therefore, no offset. • The module, and when applicable the library. In this example, the module <code>read.o</code> and the library <code>iolib.a</code>. • The section number and section name. In this example, section number <code>6</code> with the name <code>.text</code>. • The offset, specified in number of bytes, in the section. In this example, <code>0x0</code>.

Table 10: Description of a relocation error (Continued)

Possible solutions

In this case, the distance from the instruction in `myfunc` to `__read` is too long for the branch instruction.

Possible solutions include ensuring that the two `.text` sections are allocated closer to each other or using some other calling mechanism that can reach the required distance. It is also possible that the referring function tried to refer to the wrong target and that this caused the range error.

Different range errors have different solutions. Usually, the solution is a variant of the ones presented above, in other words modifying either the code or the section placement.

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. In addition to these, you can define your own that you can use to ensure that incompatible modules are not used together.

Note: In addition to the predefined attributes, compatibility is also checked against the RL78 ABI runtime attributes. These attributes deal mainly with object code compatibility, etc. They reflect compilation settings and are not user-configurable.

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. In general, 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.

Note: For IAR predefined runtime model attributes, the linker checks them in several ways.

Example

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

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

Table 11: 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", "model"
```

Note: Key names that start with two underscores are reserved by the compiler. For more information about the syntax, see *rtmodel*, page 373 and the *IAR Assembler User Guide for RL78*.

At link time, the IAR ILINK 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.

Linker optimizations

This section contains information about:

- *Virtual function elimination*, page 119
- *Small function inlining*, page 120
- *Duplicate section merging*, page 120

VIRTUAL FUNCTION ELIMINATION

Virtual Function Elimination (VFE) is a linker optimization that removes unneeded virtual functions and dynamic runtime type information.

In order for Virtual Function Elimination to work, all relevant modules must provide information about virtual function table layout, which virtual functions are called, and for which classes dynamic runtime type information is needed. If one or more modules do not provide this information, a warning is generated by the linker and Virtual Function Elimination is not performed.



If you know that modules that lack such information do not perform any virtual function calls and do not define any virtual function tables, you can use the `--vfe=forced` linker option to enable Virtual Function Elimination anyway.



In the IDE, select **Project>Options>Linker>Optimizations>Perform C++ Virtual Function Elimination** to enable this optimization.

Currently, tools from IAR Systems provide the information needed for Virtual Function Elimination in a way that the linker can use.

Note: You can disable Virtual Function Elimination entirely by using the `--no_vfe` linker option. In this case, no warning will be issued for modules that lack VFE information.

For more information, see `--vfe`, page 318 and `--no_vfe`, page 310.

SMALL FUNCTION INLINING

Small function inlining is a linker optimization that replaces some calls to small functions with the body of the function. This requires the body to fit in the space of the instruction that calls the function.



In the IDE, select **Project>Options>Linker>Optimizations>Inline small routines** to enable this optimization.



Use the linker option `--inline`.

DUPLICATE SECTION MERGING

The linker can detect read-only sections with identical contents and keep only one copy of each such section, redirecting all references to any of the duplicate sections to the retained section.



In the IDE, select **Project>Options>Linker>Optimizations>Merge duplicate sections** to enable this optimization.



Use the linker option `--merge_duplicate_sections`.

Note: This optimization can cause different functions or constants to have the same address, so if your application depends on the addresses being different, for example, by using the addresses as keys into a table, you should not enable this optimization.

The DLIB runtime environment

- Introduction to the runtime environment
- Setting up the runtime environment
- Additional information on the runtime environment
- Managing a multithreaded 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 121
- *Briefly about input and output (I/O)*, page 122
- *Briefly about C-SPY emulated I/O*, page 123
- *Briefly about retargeting*, page 124

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

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

The runtime library is delivered both as prebuilt libraries 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 131. You can find the libraries in the product subdirectories `r178\lib` and `r178\src\lib`, respectively.

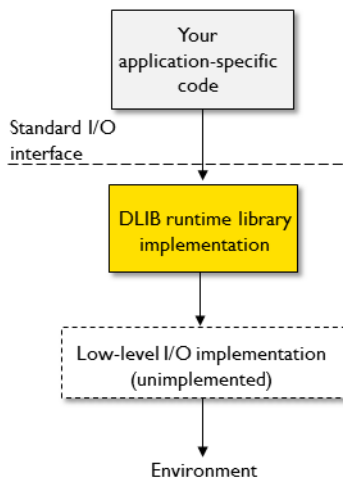
For more information about the library, see the chapter *C/C++ standard library functions*.

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:



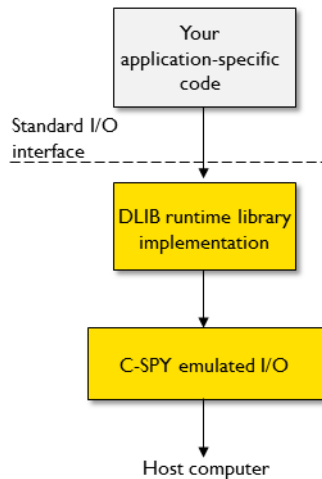
To make the standard I/O interface work, you can:

- Let the C-SPY debugger emulate I/O operations on the host computer, see *Briefly about C-SPY emulated I/O*, page 123
- *Retarget* the standard I/O interface to your target system by providing a suitable implementation of the interface, see *Briefly about retargeting*, page 124.

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
- The C-SPY debugger notifies when the application terminates or an assert fails.

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.

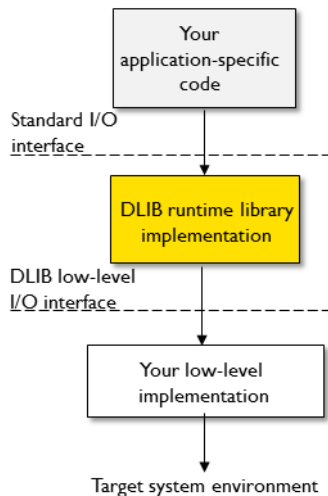
See *Setting up your runtime environment*, page 125 and *The C-SPY emulated I/O mechanism*, page 138.

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 128. See also *The DLIB low-level I/O interface*, page 145 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 125
A runtime environment with basic project settings to be used during the initial phase of development.
- *Retargeting—Adapting for your target system*, page 127
- *Overriding library modules*, page 128
- *Customizing and building your own runtime library*, page 129

See also:

- *Managing a multithreaded environment*, page 157 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 configuration* to use. Typically, choose **Tiny**, **Normal**, or **Full**.
For information about the various library configurations, see *Runtime library configurations*, page 131.

- 3 On the **Library Options** page, select **Auto with multibyte support** or **Auto without multibyte support** 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 136 and *Formatters for scanf*, page 137, respectively.
- 4 To enable C-SPY emulated I/O, choose **Project>Options>Linker>Library** and select **Include C-SPY debugging support**. See *Briefly about C-SPY emulated I/O*, page 123.



On the command line, use the linker option `--debug_lib`.

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 RL78*.

Note: If you enable debug information before compiling, this information will be included also in the linker output, unless you use the linker option `--strip`.

- 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: This function uses about 80 bytes of RAM memory.



To use this feature in the IDE, choose **Project>Options>Linker>Library** and select the option **Buffered write**.



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

```
--redirect __write=__write_buffered
```

- 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 139.

- 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 131.

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 sections are initialized. See *System startup and termination*, page 141 and *System initialization*, page 144.

Note: 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 124.

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 403.

- Assert, see `__iar_ReportAssert`, page 149.

- 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 145.

The library files that you can override with your own versions are located in the `r178\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 128.

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 123.

- 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.

Note: By default, the `NDEBUG` symbol is defined in a Release build configuration, which means asserts will no longer be checked. For more information, see `__iar_ReportAssert`, page 149.

OVERRIDING LIBRARY MODULES

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

- I 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 `r178\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 123.

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* relevant parts of the library.

Note: Customizing and building your own runtime library requires access to the library source code, which is not available for all types of IAR Embedded Workbench licenses.

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

- 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 (`src\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:

- 1 In the IDE, choose **Project>Create New Project** and use the library project template which can be used for customizing the runtime environment configuration. There is a library template for the Full library configuration, see *Runtime library configurations*, page 131

Note: 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 61.

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 `r178\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 the `r178\config\template\project` directory—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 information about configuration symbols that you might want to customize, see:

- *Configuration symbols for file input and output*, page 154
- *Locale*, page 155
- *Strtod*, page 156
- *Managing a multithreaded environment*, page 157

- 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 for RL78*.

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**.
- 3 In the **Configuration file** text box, locate your library configuration file.
- 4 Click the **Library** tab, also in the **Linker** category. Use the **Additional libraries** text box to locate your library file.

Additional information on the runtime environment

This section gives additional information on the runtime environment:

- *Bounds checking functionality*, page 131
- *Runtime library configurations*, page 131
- *Prebuilt runtime libraries*, page 132
- *Formatters for printf*, page 136
- *Formatters for scanf*, page 137
- *The C-SPY emulated I/O mechanism*, page 138
- *Math functions*, page 139
- *System startup and termination*, page 141
- *System initialization*, page 144
- *The DLIB low-level I/O interface*, page 145
- *Configuration symbols for file input and output*, page 154
- *Locale*, page 155
- *Strtod*, page 156
- *Hardware support*, page 156

BOUNDS CHECKING FUNCTIONALITY

To enable the bounds checking functions specified in Annex K (*Bounds-checking interfaces*) of the C standard, define the preprocessor symbol `__STDC_WANT_LIB_EXT1__` to 1 prior to including any system headers. See *C bounds-checking interface*, page 402.

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)	C locale, but no locale interface, 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, and optionally multibyte characters in <code>printf</code> and <code>scanf</code> , and hexadecimal floating-point numbers in <code>strtod</code> .

Table 12: 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 129

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 file will automatically be used. See *Setting up the runtime environment*, page 125.

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 265.

The prebuilt libraries are based on the default configurations, see *Runtime library configurations*, page 131.

- 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 129.

PREBUILT RUNTIME LIBRARIES

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

- Code model
- Data model
- Processor core
- Size of the `double` floating-point type
- Calling convention

- Library configuration—Normal or Full
- Use of the `--generate_far_runtime_library_calls` option.

The linker will automatically include the correct library object file and library configuration file. To explicitly specify a library configuration, use the `--dlib_config` compiler option.

Library filename syntax

The names of the libraries are constructed from these elements:

<code>{library}</code>	d1 for the IAR DLIB runtime environment
<code>{cpu}</code>	r1 for the RL78 microcontroller
<code>{code_model}</code>	Specifies the code model: n = Near f = Far
<code>{data_model}</code>	Specifies the data model: n = Near f = Far h = Huge
<code>{calling_c}</code>	Specifies the calling convention: 1 = the V1 calling convention 2 = the V2 calling convention
<code>{core}</code>	Specifies the core: 2 = S1 or S2 3 = S3
<code>{lib_config}</code>	Specifies the library configuration: n = Normal f = Full
<code>[far_rt_calls]</code>	Specifies whether the compiler option <code>--generate_far_runtime_library_calls</code> is used: f = the option is used empty if the option is not used
<code>{debug_io}</code>	Specifies the support for C-SPY emulated I/O: d = support for C-SPY emulated I/O n = no support for C-SPY emulated I/O

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

You can find the library files and the library configuration files in the directory `r178\lib`.

Groups of library files

The libraries are delivered in groups of library functions:

Library files for C/C++ standard library functions

These are the functions defined by Standard C and C++, for example functions like `printf` and `scanf`.

The names of the library files are constructed in the following way:

```
dl{cpu}{code_model}{data_model}{size_of_double}{calling_c}{core}{
lib_config}[far_rt_calls].a
```

which more specifically means

```
dlr1{n|f}{n|f|h}{f|d}{1|2}{2|3}{n|f}[f].a
```

Library files for thread support

These are the functions for thread support.

The names of the library files are constructed in the following way:

```
th{cpu}{code_model}{data_model}{size_of_double}{calling_c}{core}{
lib_config}[far_rt_calls].a
```

which more specifically means

```
thr1{n|f}{n|f|h}{f|d}{1|2}{2|3}{n|f}[f].a
```

Library files with support for timezone and daylight saving time functionality

These are the functions with support for timezone and daylight saving time functionality.

The names of the library files are constructed in the following way:

```
tz{cpu}{code_model}{data_model}{size_of_double}{calling_c}{core}{
lib_config}[far_rt_calls].a
```

which more specifically means

```
tzr1{n|f}{n|f|h}{f|d}{1|2}{2|3}{n|f}[f].a
```

Library files for C-SPY emulated I/O

These are functions for C-SPY emulated I/O.

The names of the library files are constructed in the following way:

```
dbg{cpu}{code_model}{data_model}{size_of_double}{calling_c}{core}{
debug_io}.a
```

which more specifically means

```
dbgr1{n|f}{n|f|h}{f|d}{1|2}{2|3}{d|n}.a
```

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

Table 13: Formatters for `printf`

† NoMb means without multibytes.

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 Use one of these ILINK command line options:

```
--redirect __Printf=__PrintfFull
--redirect __Printf=__PrintfFullNoMb
--redirect __Printf=__PrintfLarge
--redirect __Printf=__PrintfLargeNoMb
--redirect __Printf=__PrintfSmall
--redirect __Printf=__PrintfSmallNoMb
--redirect __Printf=__PrintfTiny
--redirect __Printf=__PrintfTinyNoMb
```

If the compiler does not recognize multibyte support, you can enable it:



Select **Project>Options>General Options>Library Options 1>Enable multibyte support**.



Use the linker option `--printf_multibytes`.

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/	Large/	Full/
	SmallNoMb†	LargeNoMb†	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
<code>wchar_t</code> support	No	No	Yes

Table 14: Formatters for `scanf`

† NoMb means without multibytes.

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 Use one of these ILINK command line options:

```
--redirect __Scanf=__ScanfFull
--redirect __Scanf=__ScanfFullNoMb
--redirect __Scanf=__ScanfLarge
--redirect __Scanf=__ScanfLargeNoMb
--redirect __Scanf=__ScanfSmall
--redirect __Scanf=__ScanfSmallNoMb
```

If the compiler does not recognize multibyte support, you can enable it:



Select **Project>Options>General Options>Library Options 1>Enable multibyte support**.



Use the linker option `--scanf_multibytes`.

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 123.

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`, `pow`, `sin`, and `tan` 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:

1 Choose **Project>Options>General Options>Library Options 1>Math functions** and choose which set to use.

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



To specify individual math functions from the command line:

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

```
--redirect _sin=__iar_sin_small
--redirect _cos=__iar_cos_small
--redirect _tan=__iar_tan_small
--redirect _log=__iar_log_small
--redirect _log2=__iar_log2_small
--redirect _log10=__iar_log10_small
--redirect _exp=__iar_exp_small
--redirect _pow=__iar_pow_small
```

```

--redirect _sinf=__iar_sin_smallf
--redirect _cosf=__iar_cos_smallf
--redirect _tanf=__iar_tan_smallf
--redirect _logf=__iar_log_smallf
--redirect _log2f=__iar_log2_smallf
--redirect _log10f=__iar_log10_smallf
--redirect _expf=__iar_exp_smallf
--redirect _powf=__iar_pow_smallf

--redirect _sinl=__iar_sin_smalll
--redirect _cosl=__iar_cos_smalll
--redirect _tanl=__iar_tan_smalll
--redirect _logl=__iar_log_smalll
--redirect _log2l=__iar_log2_smalll
--redirect _log10l=__iar_log10_smalll
--redirect _expl=__iar_exp_smalll
--redirect _powl=__iar_pow_smalll

```

More accurate versions

The functions `cos`, `pow`, `sin`, and `tan` 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:

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

```

--redirect _sin=__iar_sin_accurate
--redirect _cos=__iar_cos_accurate
--redirect _tan=__iar_tan_accurate
--redirect _pow=__iar_pow_accurate

--redirect _sinf=__iar_sin_accuratef
--redirect _cosf=__iar_cos_accuratef
--redirect _tanf=__iar_tan_accuratef
--redirect _powf=__iar_pow_accuratef

--redirect _sinl=__iar_sin_accuratel
--redirect _cosl=__iar_cos_accuratel
--redirect _tanl=__iar_tan_accuratel
--redirect _powl=__iar_pow_accuratel

```

SYSTEM STARTUP AND TERMINATION

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

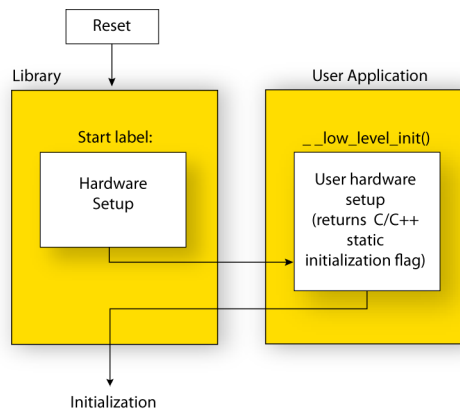
The code for handling startup and termination is located in the source files `cstartup.s`, `cexit.s`, and `low_level_init.c` located in the `r178\src\lib` directory.

For information about how to customize the system startup code, see *System initialization*, page 144.

System startup

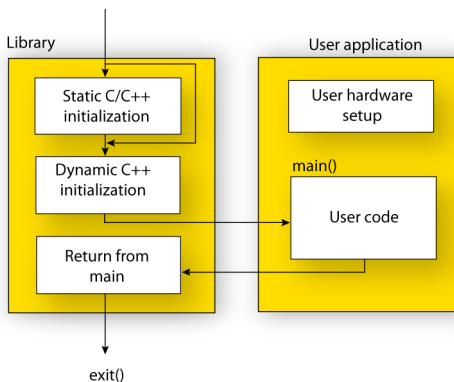
During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

For the hardware initialization, it looks like this:



- When the CPU is reset it will start executing at the program entry label `__iar_program_start` in the system startup code.
- The stack pointer is initialized
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

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

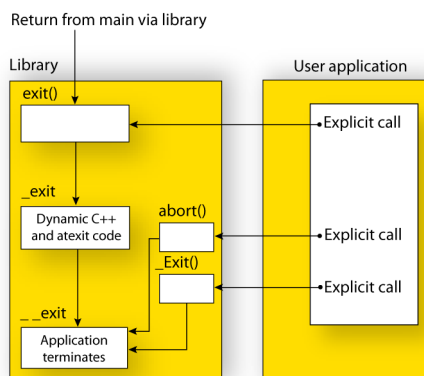


- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialization at system startup*, page 94.
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

For information about the initialization phase, see *Application execution—an overview*, page 56.

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`. See also *Setting up the atexit limit*, page 110.
- Close all open files
- Call `__exit`
- When `__exit` is reached, stop the system.

An application can also exit by calling the `abort`, the `_Exit`, or the `quick_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. The

`quick_exit` function is equivalent to the `_Exit` function, except that it calls each function passed to `at_quick_exit` before calling `__exit`.

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 `r178\src\lib` directory. See *Overriding library modules*, page 128.

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 123.

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 sections 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 `cstartup.s` file before the data sections are initialized. Modifying the `cstartup.s` file directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s` and `low_level_init.c`, located in the `r178\src\lib` directory.

Note that normally, you do not need to customize `cexit.s`.

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

Customizing `__low_level_init`

A skeleton low-level initialization file is supplied with the product: `low_level_init.c`, which is part of the prebuilt runtime environment.

Note: Static initialized variables cannot be used within the file, because variable initialization has not been performed at this point.

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

Modifying the `cstartup` file

As noted earlier, you should not modify the `cstartup.s` file if implementing your own version of `__low_level_init` is enough for your needs. However, if you do need to modify the `cstartup.s` file, 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 128.

Note: You must make sure that the linker uses the start label used in your version of `cstartup.s`. For information about how to change the start label used by the linker, see *--entry*, page 301.

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, see *Briefly about input and output (I/O)*, page 122.

These are the functions in the DLIB low-level I/O interface:

```

abort
clock
__close
__exit
getenv
__getzone
__iar_ReportAssert
__lseek
__open
raise
__read
remove
rename
signal
system

```

`__time32, __time64``__write`

Note: You should normally 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 calls that function via a standard library function, the linker 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 123.

abort

Source file	<code>r178\src\lib\runtime\abort.c</code>
Declared in	<code>stdlib.h</code>
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 124 <i>System termination</i> , page 143.

clock

Source file	<code>r178\src\lib\time\clock.c</code>
Declared in	<code>time.h</code>
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 *Briefly about retargeting*, page 124.

__close

Source file `r178\src\lib\file\close.c`

Declared in `LowLevelIOInterface.h`

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 124.

__exit

Source file `r178\src\lib\runtime__exit.c`

Declared in `LowLevelIOInterface.h`

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 124
System termination, page 143.

getenv

Source file `r178\src\lib\runtime\getenv.c`
`r178\src\lib\runtime\environ.c`

Declared in `Stdlib.h` and `LowLevelIOInterface.h`

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 124.

__getzone

Source file `r178\src\lib\time\getzone.c`

Declared in `LowLevelIOInterface.h`

Description Low-level function that returns the current time zone.

Note: You must enable the time zone functionality in the library by using the linker option `--timezone_lib`.


C-SPY debug action Not applicable.

Default implementation Returns " : ".

See also *Briefly about retargeting*, page 124 and `--timezone_lib`, page 316.

For more information, see the source file `getzone.c`.

__iar_ReportAssert

Source file	<code>r178\src\lib\runtime\xreportassert.c</code>
Declared in	<code>assert.h</code>
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 <code>__iar_ReportAssert</code> . By default, it prints an error message and calls <code>abort</code> . 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 <code>assert.h</code> . To turn off assertions, define the symbol <code>NDEBUG</code> .  In the IDE, the symbol <code>NDEBUG</code> is by default defined in a Release project and <i>not</i> defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See <i>NDEBUG</i> , page 391.
See also	<i>Briefly about retargeting</i> , page 124.

__lseek

Source file	<code>r178\src\lib\file\lseek.c</code>
Declared in	<code>LowLevelIOInterface.h</code>
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 124.

__open

Source file	<code>r178\src\lib\file\open.c</code>
Declared in	<code>LowLevelIOInterface.h</code>

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 124.

raise

Source file	<code>r178\src\lib\runtime\raise.c</code>
Declared in	<code>signal.h</code>
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 124.

__read

Source file	<code>r178\src\lib\file\read.c</code>
Declared in	<code>LowLevelIOInterface.h</code>
Description	Low-level function that reads characters from <code>stdin</code> and from files.
C-SPY debug action	Directs <code>stdin</code> to the Terminal I/O window. All other files will read the associated host file.
Default implementation	None.
See also	<i>Briefly about retargeting</i> , page 124.

Example

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at 0xFFF72:

```
#include <stddef.h>

__no_init volatile __sfr unsigned char kbIO @ 0xFFF72;

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

    /* Check for stdin
       (only necessary if FILE descriptors are enabled) */
    if (handle != 0)
    {
        return -1;
    }

    for (/*Empty*/; bufSize > 0; --bufSize)
    {
        unsigned char c = kbIO;
        if (c == 0)
            break;

        *buf++ = c;
        ++nChars;
    }

    return nChars;
}
```

For information about the handles associated with the streams, see *Retargeting—Adapting for your target system*, page 127.

For information about the @ operator, see *Controlling data and function placement in memory*, page 222.

remove

Source file	r178\src\lib\file\remove.c
Declared in	stdio.h
Description	Standard C library function that removes a file.

C-SPY debug action	Writes a message to the Debug Log window and returns -1.
Default implementation	Returns 0 to indicate success, but without removing a file.
See also	<i>Briefly about retargeting</i> , page 124.

rename

Source file	<code>r178\src\lib\file\rename.c</code>
Declared in	<code>stdio.h</code>
Description	Standard C library function that renames a file.
C-SPY debug action	None.
Default implementation	Returns -1 to indicate failure.
See also	<i>Briefly about retargeting</i> , page 124.

signal

Source file	<code>r178\src\lib\runtime\signal.c</code>
Declared in	<code>signal.h</code>
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 124.

system

Source file	<code>r178\src\lib\runtime\system.c</code>
Declared in	<code>stdlib.h</code>

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	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 124.

__time32, __time64

Source file	<code>r178\src\lib\time\time.c</code> <code>r178\src\lib\time\time64.c</code>
Declared in	<code>time.h</code>
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 124.

__write

Source file	<code>r178\src\lib\file\write.c</code>
Declared in	<code>LowLevelIOInterface.h</code>
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 Terminal I/O window. All other files will write to the associated host file.
Default implementation	None.
See also	<i>Briefly about retargeting</i> , page 124.

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 0xFFFF70:

```
#include <stddef.h>

__no_init volatile __sfr unsigned char lcdIO @ 0xFFFF70;

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

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

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

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

    return nChars;
}
```

For information about the handles associated with the streams, see *Retargeting—Adapting for your target system*, page 127.

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 131, 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 129.

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 levels 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 131.

The DLIB runtime library can be used in two main modes:

- Using a full library configuration that has a locale interface, which makes it possible to switch between different locales during runtime

The application starts with the C locale. To use another locale, you must call the `setlocale` function or use the corresponding mechanisms in C++. The locales that the application can use are set up at linkage.
- Using a normal library configuration that does not have a locale interface, where the C locale is hardwired into the application.

Note: If multibytes are to be printed, you must make sure that the implementation of `__write` in the DLIB low-level I/O interface can handle them.

Specifying which locales that should be available in your application



Choose **Project>Options>General Options>Library Options 2>Locale support**.



Use the linker option `--keep` with the tag of the locale as the parameter, for example:

```
--keep _Locale_cs_CZ_iso8859_2
```

The available locales are listed in the file `SupportedLocales.json` in the `r178\config` directory, for example:

```
['Czech language locale for Czech Republic', 'iso8859-2',
'cs_CZ.iso8859-2', '_Locale_cs_CZ_iso8859_2'],
```

The line contains the full locale name, the encoding for the locale, the abbreviated locale name, and the tag to be used as parameter to the linker option `--keep`.

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 available encodings are ISO-8859-1, ISO-8859-2, ISO-8859-4, ISO-8859-5, ISO-8859-7, ISO-8859-8, ISO-8859-9, ISO-8859-15, CP932, and UTF-8.

For a complete list of the available locales and their respective encoding, see the file `SupportedLocales.json` in the `rl78\config` directory.

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");
```

STRTOD

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

- 1 Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.
- 2 Rebuild the library, see *Customizing and building your own runtime library*, page 129.

HARDWARE SUPPORT

Some RL78 microcontrollers have a hardware multiplier/divider unit. To include runtime support for this unit, use the sample library code provided in the `rl78\src\lib\hw_multiply_division_units` directory.



In the IDE, select the **Use Hardware Multiplier/Divider Unit** option on the **Project Options>General Options>Library Configuration** page. If the currently selected RL78 device does not have a hardware multiplier/divider unit, this option will not be available.

Managing a multithreaded environment

This section contains information about:

- *Multithread support in the DLIB runtime environment*, page 157
- *Enabling multithread support*, page 158

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 C 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).
- C++ dynamically initialized function-local objects with static storage duration.
- C++ locale facet handling
- C++ regular expression handling
- C++ terminate and unexpected handling

These library objects use TLS:

Library objects using TLS	When these functions are used
Error functions	<code>errno</code> , <code>strerror</code>

Table 15: 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 C++ is used in a 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

To configure multithread support for use with threaded applications:

- 1 To enable multithread support:



On the command line, use the linker option `--threaded_lib`.

If C++ is used, the compiler option `--guard_calls` should be used as well to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.



In the IDE, choose **Project>Options>General Options>Library Configuration>Enable thread support in the library**. This will invoke the linker option `--threaded_lib` and if C++ 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.
 - If file streams are used, implement code for the library's file stream locks interface.
 - Implement code that handles thread creation, thread destruction, and TLS access methods for the library.

You can find the required declaration of functions in the `DLib_Threads.h` file. There you will also find more information.

- 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.

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 RL78 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 useful in, for example, time-critical routines.

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

The advantage of an intrinsic function compared to using inline assembler is that the compiler has all necessary information to interface the sequence properly with register

allocation and variables. The compiler also knows how to optimize functions with such sequences; something the compiler is unable to do with inline assembler sequences. The result is that you get the desired sequence properly integrated in your code, and that the compiler can optimize the result.

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

MIXING C AND ASSEMBLER MODULES

It is possible to write parts of your application in assembler and mix them with your C or C++ modules.

This gives several benefits compared to using inline assembler:

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

This causes some overhead in the form of a `CALL` and a `RET` instruction, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all registers are destroyed by an inline assembler instruction. In many cases, the overhead of the extra instructions can be removed by the optimizer.

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

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

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

The first question is discussed in the section *Calling assembler routines from C*, page 163. The following two are covered in the section *Calling convention*, page 166.

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 177.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 163, and *Calling assembler routines from C++*, page 165, respectively.

Note: To comply with the RL78 ABI, the compiler generates assembler labels for symbol and function names by prefixing an underscore. You must remember to add this extra underscore when you access C symbols from assembler. For example, `main` must be written as `_main`.

Similarly, when referencing an external assembly module from C, an underscore will be added to the symbol used in the C module, so the name of the assembly module must start with an added underscore.

INLINE ASSEMBLER

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

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

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

The syntax is:

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

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

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

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

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

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

static __near char sFlag;

void Foo(void)
{
    while (!sFlag)
    {
        asm("MOV A, S:UART1_SR");
        asm("MOV sFlag, A");
    }
}
```

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

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

Inline assembler sequences have no well-defined interface with the surrounding code generated from your C or C++ code. This makes the inline assembler code fragile, and might also become a maintenance problem if you upgrade the compiler in the future. There are also several limitations to using inline assembler:

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Alignment cannot be controlled; this means, for example, that `DC16` directives might be misaligned
- Auto variables cannot be accessed.
- Labels cannot be declared.

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

Calling assembler routines from C

An assembler routine that will be called from C must:

- Conform to the calling convention
- Have a `PUBLIC` entry-point label
- Be declared as external before any call, to allow type checking and optional promotion of parameters, as in these examples:

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

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

The compiler supports two different calling conventions, see *Calling convention*, page 166.

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: 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:

```
iccrl78 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 `s`. 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.s`.

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, to exclude the CFI directives from the list file, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**.



On the command line, to exclude the CFI directives from the list file, use the option `-lB` instead of `-lA`.

Note: 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 177.

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);
}
```

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);
    }
};
```

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—one old, which is used in version 1.x of the compiler, and one new which is the default. This section describes the calling conventions used by the compiler. These items are examined:

- Choosing a calling convention
- Function declarations
- C and C++ linkage
- Registers and bank switching
- Preserved versus scratch registers
- Function entrance
- Function exit
- Return address handling
- Bitfield allocation

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

Note: The V2 calling convention complies with the RL78 ABI standard.

CHOOSING A CALLING CONVENTION

You can choose between these calling conventions:

- The V1 calling convention is compatible with the IAR C/C++ Compiler for RL78 version 1.40.x except for how stack arguments are cleaned up. This calling convention is recommended if you have assembler code written for an earlier version of the compiler. To make an individual function use this calling convention, use the `__v1_call` keyword, for example:

```
extern
__v1_call void doit(int arg);
```

- The V2 calling convention complies with the RL78 ABI and is recommended when you link with code produced by a tool from another vendor. To make an individual function use this calling convention, use the `__v2_call` keyword.

The V2 calling convention is used by default. To make the compiler use the V1 calling convention instead, use the `--calling_convention` command line option.

In the IDE, choose the calling convention on the **Project>General Options>Target** page.

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
```

REGISTERS AND BANK SWITCHING

The compiler only supports switching register banks for interrupt functions, to avoid having to save all registers. Any register banks that you have not used for bank switching, you can use in your assembler source code.

PRESERVED VERSUS SCRATCH REGISTERS

The general RL78 CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

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

The following registers are used as scratch registers:

- The registers `AX`, `HL`, `CS` and `ES`. In the `V2` calling convention, the `BC` and `DE` registers are also scratch registers.
- Registers that are used as register parameters and for returning values by a function.

Preserved registers

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

- In the `V1` calling convention, the registers `BC` and `DE` are preserved registers.
- In the `V2` calling convention, no registers are preserved across non-interrupt function calls. Across interrupts, the registers `AX`, `HL`, `BC`, and `DE` are preserved in the active register bank, and the system registers `PC` and `PSW`.

Only registers in the current register bank are preserved. The compiler does not use the other register banks unless you have specified that it shall.

Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer register must at all times point to or below the last element on the stack. If an interrupt occurs, everything below the point the stack pointer points to will be destroyed.

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 conventions are 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. In the V1 calling convention, the parameters are also passed on the stack in these cases:

- Structure types: `struct`, `union`, and classes, except structs of sizes 1, 2, and 4 (with a minimum alignment of 2 if the size is 2 or 4)
- Unnamed parameters to variable length (variadic) functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

This means, basically, that objects greater than 4 bytes are passed on the stack and, in the V1 calling convention, all 3-byte objects. For more details, see *Register parameters*, page 169.

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, a pointer to the allocated space is passed as a hidden parameter. Hidden parameters are treated as ordinary parameters.

Register parameters

The registers available for passing parameters are AX, BC, and DE.

Parameters	Passed in registers V1 calling convention	Passed in registers V2 calling convention
8-bit values	A, B, C, X, D, E	A, B, C, X, D, E
16-bit values	AX, BC, DE	AX, BC, DE
24-bit values	Passed on the stack	C : AX, X : BC, E : BC, X : DE, B : DE (struct) A : DE, X : DE, C : DE, B : DE, X : BC (far pointer)
32-bit values	BC : AX	BC : AX, DE : BC
64-bit values	Passed on the stack	Passed on the stack

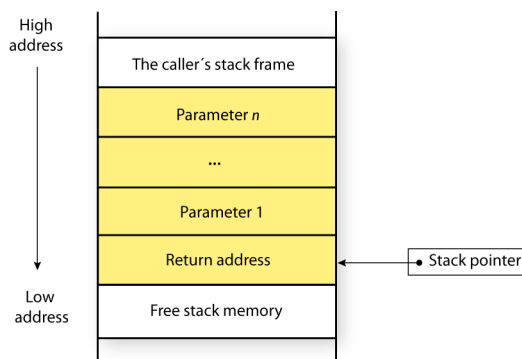
Table 16: Registers used for passing parameters

The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order from left to right, the first parameter is assigned to the available register or registers. Should there be no suitable register available, the parameter is passed on the stack.

Stack parameters and layout

Stack parameters are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer (toward low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack that is divisible by two, etc. Objects on the stack are stored with alignment 2, which means that 1-byte objects (`char`) will occupy 2 bytes.

This figure illustrates how parameters are stored on the stack:



FUNCTION EXIT

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

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

Registers used for returning values

The registers available for returning values are `A`, `B`, `C`, and `X`.

Return values	Passed in registers V1 calling convention	Passed in registers V2 calling convention
8-bit values	A	A
16-bit values	AX	AX
24-bit values	A : HL	C : AX (struct) A : DE (far pointer)
32-bit values	BC : AX	BC : AX

Table 17: Registers used for returning values

Return values	Passed in registers V1 calling convention	Passed in registers V2 calling convention
64-bit values	Passed via a pointer	Passed via a pointer

Table 17: Registers used for returning values (Continued)

Stack layout at function exit

It is the responsibility of the calling function to clean the stack before function exit, in other words, responsible for cleaning the pushed parameters before the function returns.

Return address handling

A function written in assembler language should, when finished, return to the caller. At a function call, the return address is stored on the stack, for example:

```
CALL    func
```

Typically, a function returns by using the `RET` instruction.

There are two exceptions to this rule:

- For interrupt functions, the instruction `RETI` is used for a function return
- For the break interrupt, `BRK` instruction, the instruction `RETB` is used for a function return.

BITFIELD ALLOCATION

The two calling conventions use different bitfield allocation strategies:

- If the V1 calling convention is the default calling convention, bitfield containers of different types cannot overlap in memory (*disjoint types* allocation)
- If the V2 calling convention is the default calling convention, you can choose between disjoint types allocation and *joined types* allocation, where each bitfield is placed in the next container of its base type that has enough available bits to accommodate the bitfield. This allocation strategy is used by default if the V2 calling convention is the default calling convention.

Note that overriding the calling convention for a single function by use of an extended keyword does not affect the bitfield allocation strategy for that function; it is still controlled by the default calling convention for the project. For more information about bitfield allocation, see *Bitfields*, page 324.

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

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

```

                name    return
                section CODE:CODE
                public  _add1
add1:
                incw    ax
                ret
                end

```

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 `AX`. The return value is passed back to its caller in the register `AX`.

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 and pass a pointer to it as a hidden first parameter. If you are using the near data model, the pointer to the location where the return value should be stored is passed in `AX`. The caller assumes that these registers remain untouched. The parameter `x` is passed in `BC`.

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 `AX`, and the return value is returned in `AX`.

Assembler instructions used for calling functions

This section presents the assembler instructions that can be used for calling and returning from functions on the RL78 microcontroller.

Functions can be called in different ways—directly, or via a function pointer, or via a function pointer table. In this section we will discuss how these types of calls will be performed for each code model.

The normal function calling instruction is the `call` instruction:

```
call label
```

The following sections illustrates how the different code models perform function calls.

CALLING FUNCTIONS IN THE NEAR CODE MODEL

A direct call using this code model is simply:

```
call n:label16
```

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

```
name    callFuncPtr
rseg    CODE:CODE
extern  funcPtr

movw    ax, n:funcPtr ; Load function address
mov     cs, #0        ; clear byte 3
call    ax           ; Make function call
end
```

The address is stored in a register and is then used for calling the function. Calls via a function pointer reach the whole 32-bit address space.

CALLING FUNCTIONS IN THE FAR CODE MODEL

A direct call using this code model is simply:

```
call f:label24
```

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

```
name    callFuncPtr
rseg    CODE:CODE
extern  funcPtr

mov     a, n:funcPtr+2 ; Load byte3 of function
                           ; address

mov     cs, A
movw   ax, n:funcPtr ; Load function address
call   ax             ; Make function call
end
```

The address is stored in a register and is then used for calling the function. Calls via a function pointer reach the whole 32-bit address space.

MAKING FUNCTION TABLE CALLS

A function table call is simply:

```
callt [index]
```

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

```
name    callTable
rseg    CLTVEC:CODE:REORDER:NOROOT(1)

__T_func:
dw      func
rseg    CODE:CODE
extern  func

callt   [__T_func]           ; Make function call
end
```

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 RL78 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 SADDR MEMORY ACCESS METHOD

Saddr (short addressable area) memory is a 256-bytes area located in near memory. It can be accessed using an 8-bit address or via a near pointer.

Examples

These examples access saddr memory in different ways:

```
mov    a, s:x           ; Access the global variable x.

mov    hl, #y           ; Access an entry in the
addw   ax, hl           ; global array y.
movw   hl, ax
mov    a, [hl]

mov    a, [hl+2]       ; Access through a pointer.
```

THE NEAR MEMORY ACCESS METHOD

Near memory is located in the last 64 Kbytes of memory.

Examples

These examples access near memory in different ways:

```

mov    a, n:x           ; Access the global variable x.

mov    a, (y & 0xffff)[bc] ; Access an entry in the
                           ; global array y.

mov    a, [hl+2]       ; Access through a pointer.

```

THE FAR MEMORY ACCESS METHOD

Far memory is the entire 1 Mbyte memory area with the constraint the each object must fit within a 64 Kbytes page.

Examples

These examples access near memory in different ways:

```

mov    es, byte3(x)     ; Access the global variable x.
mov    a, es:x

addw   ax, #lwrđ(y)     ; Access an entry in the
movw   hl, ax           ; global array y.
mov    es, byte3(y)
mov    a, es:[hl]

mov    a, es:[hl+2]     ; Access through a pointer.

```

THE HUGE ACCESS METHOD

The huge method accesses the same memory range as the far method, but without the 64 Kbytes restriction. It is much slower than far accesses if the object is so large that more than one read/write operation is required.

Examples

These examples make huge accesses to memory in different ways:

```

mov     es, byte3(x)    ; Access the global variable x.
mov     a, es:x

addw   ax, #lwrđ(y)    ; Access an entry in the
movw   hl, ax          ; global array y.
mov     a, c
addc   a, #byte3(y)
mov     es, a
mov     a, es:[hl]

mov     a, es:[hl]     ; Access through a pointer.

```

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 RL78*.

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 regular stack and of the interrupt stack, respectively
A, B, C, D, E, H, L, X	Normal registers
CS_REG	Bit 16–19 in the code address space
ES_REG	Bit 16–19 in the data address space
MACRH	The high 16 bits of the MAC register
MACRL	The low 16 bits of the MAC register
SP	The stack pointer register
?SPH	The upper four bits of ?SP20 (always 0xF)
?RET	For return addresses (a 20-bit address)
W0–W127	The wrkseg area

Table 18: 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 (the file has been trimmed down for reasons of space):

```

NAME Cfi

RTMODEL "__SystemLibrary", "DLib"
RTMODEL "__calling_convention", "iar"
RTMODEL "__code_model", "near"
RTMODEL "__core", "r178_2"
RTMODEL "__data_model", "near"
RTMODEL "__double_size", "32"
RTMODEL "__far_rt_calls", "false"
RTMODEL "__rt_version", "1"

#define SHT_PROGBITS 0x1

EXTERN F

PUBLIC cfiExample

CFI Names cfiNames0
CFI StackFrame CFA SP NEARDATA
CFI Resource A:8, X:8, B:8, C:8, D:8, E:8, H:8, L:8,
        CS_REG:4, ES_REG:4
CFI VirtualResource ?RET:20
CFI Resource MACRH:16, MACRL:16, W0:8, W1:8, W2:8,
        W3:8, W4:8, W5:8
CFI Resource W6:8, W7:8, W8:8, W9:8, W10:8, W11:8,
        W12:8, W13:8, W14:8
CFI Resource W15:8, W16:8, W17:8, W18:8, W19:8, W20:8,
        W21:8, W22:8
CFI Resource W23:8, W24:8, W25:8, W26:8, W27:8, W28:8,
        W29:8, W30:8
CFI Resource W31:8, W32:8, W33:8, W34:8, W35:8, W36:8,
        W37:8, W38:8
CFI Resource W39:8, W40:8, W41:8, W42:8, W43:8, W44:8,
        W45:8, W46:8
CFI Resource W47:8, W48:8, W49:8, W50:8, W51:8, W52:8,
        W53:8, W54:8

```

```

CFI Resource W55:8, W56:8, W57:8, W58:8, W59:8, W60:8,
             W61:8, W62:8
CFI Resource W63:8, W64:8, W65:8, W66:8, W67:8, W68:8,
             W69:8, W70:8
CFI Resource W71:8, W72:8, W73:8, W74:8, W75:8, W76:8,
             W77:8, W78:8
CFI Resource W79:8, W80:8, W81:8, W82:8, W83:8, W84:8,
             W85:8, W86:8
CFI Resource W87:8, W88:8, W89:8, W90:8, W91:8, W92:8,
             W93:8, W94:8
CFI Resource W95:8, W96:8, W97:8, W98:8, W99:8, W100:8,
             W101:8, W102:8
CFI Resource W103:8, W104:8, W105:8, W106:8, W107:8,
             W108:8, W109:8
CFI Resource W110:8, W111:8, W112:8, W113:8, W114:8,
             W115:8, W116:8
CFI Resource W117:8, W118:8, W119:8, W120:8, W121:8,
             W122:8, W123:8
CFI Resource W124:8, W125:8, W126:8, W127:8, SP:16,
             ?SPH:4
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 1
CFI DataAlign 1
CFI ReturnAddress ?RET CODE
CFI CFA SP+4
CFI A Undefined
CFI X Undefined
CFI B SameValue
CFI C SameValue
CFI D SameValue
CFI E SameValue
CFI H Undefined
CFI L Undefined
CFI CS_REG Undefined
CFI ES_REG Undefined
CFI ?RET Frame(CFA, -4)
CFI MACRH Undefined
CFI MACRL Undefined

```

```

CFI W0 SameValue
CFI W1 SameValue
...
CFI Wn SameValue
...
CFI W126 SameValue
CFI W127 SameValue
CFI ?SPH Undefined
CFI EndCommon cfiCommon0

SECTION LOVE:CODE:NOROOT(0)
cfiExample:
CFI Block cfiBlock0 Using cfiCommon0
CFI Function cfiExample
PUSH      DE
CFI E Frame(CFA, -6)
CFI D Frame(CFA, -5)
CFI CFA SP+6
MOVW      DE, AX
CALL      _F
MOVW      HL, AX
MOVW      AX, DE
ADDW      AX, HL
POP       DE
CFI E SameValue
CFI D SameValue
CFI CFA SP+4
RET
CFI EndBlock cfiBlock0

END

```

Note: The header file `cfi.m` contains the macros `cfi_Names0` and `cfi_Common0`, 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 RL78 supports the INCITS/ISO/IEC 9899:2018 standard, also known as C18. C18 addresses defects in C11 (INCITS/ISO/IEC 9899:2012) without introducing any new language features. This means that the C11 standard is also supported. In this guide, the C18 standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

The compiler will accept source code written in the C18 standard or a superset thereof.

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.

With Standard C enabled, the IAR C/C++ Compiler for RL78 can compile all C18/C11 source code files, except for those that depend on atomic or thread-related system header files.

Annex K (*Bounds-checking interfaces*) of the C standard is supported. See *Bounds checking functionality*, page 131.

For an overview of the differences between the various versions of the C standard, see the Wikipedia articles *C18 (C standard revision)*, *C11 (C standard revision)*, or *C99*.

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 185. 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 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 159. 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 395.

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>	Strict	All IAR C language extensions are disabled—errors are issued for anything that is not part of Standard C.
None	Standard	All extensions to Standard C are enabled, but no extensions for embedded systems programming. For information about extensions, see <i>IAR C language extensions</i> , page 185.
<code>-e</code>	Standard with IAR extensions	All IAR C language extensions are enabled.

Table 19: Language extensions

* In the IDE, choose **Project>Options>C/C++ Compiler>Language 1>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 187.

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 section*

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

section. For more information about using these features, see *Controlling data and function placement in memory*, page 222, and *location*, page 368.

- *Alignment control*

Each data type has its own alignment. For more information, see *Alignment*, page 321. If you want to change the alignment, the `__packed` data type attribute, 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.

See also the Standard C file `stdalign.h`.

- *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 324.

Dedicated section operators

The compiler supports getting the start address, end address, and size for a section with these built-in section operators:

<code>__section_begin</code>	Returns the address of the first byte of the named section or block.
<code>__section_end</code>	Returns the address of the first byte <i>after</i> the named section or block.
<code>__section_size</code>	Returns the size of the named section or block in bytes.

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

These operators behave syntactically as if declared like:

```
void * __section_begin(char const * section)
void * __section_end(char const * section)
size_t __section_size(char const * section)
```

When you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined section, or when you use named blocks in the linker

configuration file, the section operators can be used for getting the start and end address of the memory range where the sections or blocks were placed.

The named *section* must be a string literal and it must have been declared earlier with the `#pragma section` directive. If the section was declared with a memory attribute *memattr*, the type of the `__section_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.

The operators are implemented in terms of *symbols* with dedicated names, and will appear in the linker map file under these names:

Operator	Symbol
<code>__section_begin(sec)</code>	<code>sec\$\$Base</code>
<code>__section_end(sec)</code>	<code>sec\$\$Limit</code>
<code>__section_size(sec)</code>	<code>sec\$\$Length</code>

Table 20: Section operators and their symbols

Note: The linker will not necessarily place sections with the same name consecutively when these operators are not used. Using one of these operators (or the equivalent symbols) will cause the linker to behave as if the sections were in a named block. This is to assure that the sections are placed consecutively, so that the operators can be assigned meaningful values. If this is in conflict with the section placement as specified in the linker configuration file, the linker will issue an error.

Example

In this example, the type of the `__section_begin` operator is `void __near *`.

```
#pragma section="MYSECTION" __near
...
section_start_address = __section_begin("MYSECTION");
```

See also *section*, page 374, and *location*, page 368.

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 329.
- 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-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;
}
```

- 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. Therefore, `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.)

- Empty translation unit
A translation unit (input file) might be empty of declarations.
- Assignment of pointer types
Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level, for example, `int **` to `const int **`. Comparisons and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- Pointers to different function types
Pointers to different function types might be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. A warning is issued. This extension is not allowed in C++ mode.
- Assembler statements
Assembler statements are accepted. This is disabled in strict C mode because it conflicts with the C standard for a call to the implicitly declared `asm` function.
- `#include_next`
The non-standard preprocessing directive `#include_next` is supported. This is a variant of the `#include` directive. It searches for the named file only in the directories on the search path that follow the directory in which the current source file (the one containing the `#include_next` directive) is found. This is an extension found in the GNU C compiler.
- `#warning`
The non-standard preprocessing directive `#warning` is supported. It is similar to the `#error` directive, but results in a warning instead of a catastrophic error when processed. This directive is not recognized in strict mode. This is an extension found in the GNU C compiler.
- Concatenating strings
Mixed string concatenations are accepted.

```
wchar_t * str="a" L "b";
```

Using C++

- Overview—Standard C++
- Enabling support for C++
- C++ feature descriptions
- C++ language extensions
- Porting code from EC++ or EEC++

Overview—Standard C++

The IAR C++ implementation fully complies with the ISO/IEC 14882:2015 C++ standard, except for source code that depends on atomic or thread-related system headers.

The ISO/IEC 14882:2015 C++ standard is also known as C++14. In this guide, this standard is referred to as Standard C++.

The IAR C/C++ compiler accepts source code written in the C++14 standard or a superset thereof.

For an overview of the differences between the various versions of the C++ standard, see the Wikipedia articles *C++17*, *C++14*, *C++11*, or *C++* (for information about C++98).

EXCEPTIONS AND RTTI

Exceptions and RTTI are not supported. Thus, the following are not allowed:

- throw expressions
- try-catch statements
- Exception specifications on function definitions
- The typeid operator
- The dynamic_cast operator

Enabling support for C++



In the compiler, the default language is C.

To compile files written in Standard C++, use the `--c++` compiler option. See `--c++`, page 259.



To enable C++ in the IDE, choose

Project>Options>C/C++ Compiler>Language 1>Language>C++.

C++ feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for RL78, 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 __saddr memory at
    // address 0xFFE30
    static __saddr __no_init int mI @ 0xFFE30;

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

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

    // Locate a virtual function in __near_func memory
    virtual __near_func 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 __far C
{
public:
    void MyF();           // Has a this pointer of type C __far *
    void MyF() const;    // Has a this pointer of type
                        // C __far const *
    C();                 // Has a this pointer pointing into far
                        // memory
    C(C const &);        // Takes a parameter of type C __far
                        // const & (also true of generated copy
                        // constructor)

    int mI;
};

C Ca;                   // Resides in far memory instead of the
                        // default memory
C __near Cb;            // Resides in near memory, the 'this'
                        // pointer still points into far memory

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

C *Cp1;                 // Creates a pointer to far memory
C __near *Cp2;          // Creates a pointer to near 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 __far 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 `__far`.

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

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

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

```

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

```

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

```

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

For more information about memory types, see *Memory types*, page 66.

TEMPLATES

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

Templates and data memory attributes

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

In 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 __far is the memory type of the default
// pointer.
template<typename> class Z {};
template<typename T> class Z<T *> {};

Z<int __near *> Zn;    // T = int __near
Z<int __far *> Zf;    // T = int
Z<int      *> Zd;    // T = int
```

In 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 __far is the memory type of the default
// pointer.
template<typename T> void fun(T *);

void MyF()
{
    fun((int __near *) 0); // T = int. The result is different
                          // than the analogous situation with
                          // class template specializations.
    fun((int      *) 0); // T = int
    fun((int __far *) 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 __far is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
    fun((int __near *) 0); // T = int __near
```

AUTO AND DECLTYPE

The C++11 keywords `auto` and `decltype` behave exactly like function template parameter deduction. That is, if deduction is made from a pointer type that can be implicitly converted to a default pointer type, the default pointer type is used instead.

Example

```
__memattr1 int var1;
__memattr2 int var2;
__memattr3 int var3;

auto p1 = &var1;    // type is "int __memattr2 *"
auto p2 = &var2;    // type is "int __memattr2 *"
auto p3 = &var3;    // type is "int __memattr3 *"

decltype(&var1) q1; // type is "int __memattr2 *"
decltype(&var2) q2; // type is "int __memattr2 *"
decltype(&var3) q3; // type is "int __memattr3 *"
```

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 attribute with an associated heap, that is, `__near`, `__far`, and `__huge`.

```
#include <stddef.h>

// Assumes that there are heaps for __near, __far, and __huge
void __far *operator new __far (__far_size_t);
void __near *operator new __near(__near_size_t);
void __huge *operator new __huge (__huge_size_t);
void operator delete(void __far *);
void operator delete(void __near *);
void operator delete(void __huge *);

// And correspondingly for array new and delete operators
void __far *operator new[] __far (__far_size_t);
void __near *operator new[] __near(__near_size_t);
void __huge *operator new[] __huge(__huge_size_t);
void operator delete[](void __far *);
void operator delete[](void __near *);
void operator delete[](void __huge *);
```

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 __near(__near_size_t)
    int __near *p = new __near int;

    // Calls operator new __near(__near_size_t)
    int __near *q = new int __near;

    // Calls operator new[] __near(__near_size_t)
    int __near *r = new __near int[10];

    // Calls operator new __far(__far_size_t)
    class __far S
    {
    };
    S *s = new S;

    // Calls operator delete(void __near *)
    delete p;
    // Calls operator delete(void __far *)
    delete s;

    int __far *t = new __near 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 141.

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.

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.

This is the same behavior as using the `nothrow` variants of `new`.

POINTER TO MEMBER FUNCTIONS

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a member function, make sure that all functions that should be pointed to reside in the default memory or a memory contained in the default memory.

Example

```
class X
{
public:
    __near_func void F();
};

void (__near_func X::*PMF)(void) = &X::F;
```

DEBUG SUPPORT IN C-SPY

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

For more information, see the *C-SPY® Debugging Guide for RL78*.

C++ language extensions

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

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

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

- 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
};
```

- It is allowed to specify an array with no size or size 0 as the last member of a struct. For example:

```
typedef struct
{
    int i;
    char ir[0]; // Zero-length array
};
```

```
typedef struct
{
    int i;
    char ir[]; // Zero-length array
};
```

- Arrays of incomplete types

An array can have an incomplete `struct`, `union`, `enum`, or `class` 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.

- Concatenating strings

Mixed string literal concatenations are accepted.

```
wchar_t * str = "a" L "b";
```

- Trailing comma

A trailing comma in the definition of an enumeration type is silently accepted.

Except where noted, all of the extensions described for C are also allowed in C++ mode.

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

Porting code from EC++ or EEC++

Apart from the fact that Standard C++ is a much larger language than EC++ or EEC++, there are two issues that might prevent EC++ and EEC++ code from compiling:

- The library is placed in `namespace std`.

There are two remedy options:

- Prefix each used library symbol with `std::`.
- Insert `using namespace std;` after the last include directive for a C++ system header file.
- Some library symbols have changed names or parameter passing.

To resolve this, look up the new names and parameter passing.

Application-related considerations

- Output format considerations
- Stack considerations
- Heap considerations
- Interaction between the tools and your application
- Checksum calculation for verifying image integrity
- Patching symbol definitions using `$$Super$$` and `$$Sub$$`

Output format considerations

The linker produces an absolute executable image in the ELF/DWARF object file format.

You can use the IAR ELF Tool—`ielftool`—to convert an absolute ELF image to a format more suitable for loading directly to memory, or burning to a PROM or flash memory etc.

`ielftool` can produce these output formats:

- Plain binary
- Motorola S-records
- Intel hex.

For a complete list of supported output formats, run `ielftool` without options.

Note: `ielftool` can also be used for other types of transformations, such as filling and calculating checksums in the absolute image.

The source code for `ielftool` is provided in the `r178/src` directory. For more information about `ielftool`, see *The IAR ELF Tool—ielftool*, page 460.

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 memory.

For more information about the stack size, see *Setting up stack memory*, page 109, and *Saving stack space and RAM memory*, page 232.

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 sections used for the heap
- Allocating the heap size, see *Setting up heap memory*, page 110.

HEAP SECTIONS IN DLIB

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

```
__near_malloc
```

If you use any of the standard functions without a prefix, the function will be mapped to the default memory type `near`.

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

For information about available heaps, see *Dynamic memory on the heap*, page 75.

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 RL78 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 `--define_symbol`. 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.
- Creating an exported configuration symbol by using the command line option `--config_def` or the configuration directive `define symbol`, and exporting the symbol using the `export symbol` directive. `ILINK` will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.

One advantage of this symbol definition is that this symbol can also be used in expressions in the configuration file, for example, to control the placement of sections into memory ranges.

- Using the compiler operators `__section_begin`, `__section_end`, or `__section_size`, or the assembler operators `SFB`, `SFE`, or `SIZEOF` on a named section or block. These operators provide access to the start address, end address, and size of a contiguous sequence of sections with the same name, or of a linker block specified in the linker configuration file.
- The command line option `--entry` 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:

```
--define_symbol NrOfElements=10
--config_def HEAP_SIZE=1024
```

The linker configuration file can look like this:

```
define memory Mem with size = 4G;
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* Export of symbol */
export symbol MY_HEAP_SIZE;

/* Setup a heap area with a size defined by an ILINK option */
define block MyHEAP with size = MY_HEAP_SIZE, alignment = 4 {};

place in RAM { block MyHEAP };
```

Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by ILINK 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 ILINK option, a symbol that in the
* configuration file was made available to the application.
*/
extern char MY_HEAP_SIZE;
```



```

/* Declare the section that contains the heap. */
#pragma section = "MYHEAP"

char *MyHeap()
{
    /* First get start of statically allocated section, */
    char *p = __section_begin("MYHEAP");

    /* ...then we zero it, using the imported size. */
    for (int i = 0; i < (int) &MY_HEAP_SIZE; ++i)
    {
        p[i] = 0;
    }
    return p;
}

```

Checksum calculation for verifying image integrity

This section contains information about checksum calculation:

- *Briefly about checksum calculation*, page 209
- *Calculating and verifying a checksum*, page 211
- *Troubleshooting checksum calculation*, page 216

For more information, see also *The IAR ELF Tool—ielftool*, page 460.

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 IAR ELF Tool—*ielftool*—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 `ieleftool` 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.

Always 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.
- 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, 4, or 8 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: 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. `ie1ftool` also provides support for controlling alignment, complement, bit order, byte order within words, 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`.

- I If you are using `ie1ftool` from the command line, you must first allocate a memory location for the calculated checksum.

Note: If you instead are using the IDE (and not the command line), the `__checksum`, `__checksum_begin`, and `__checksum_end` symbols, and the `.checksum` section are *automatically* allocated when you calculate the checksum, which means that you can skip this step.

You can allocate the memory location in two ways:

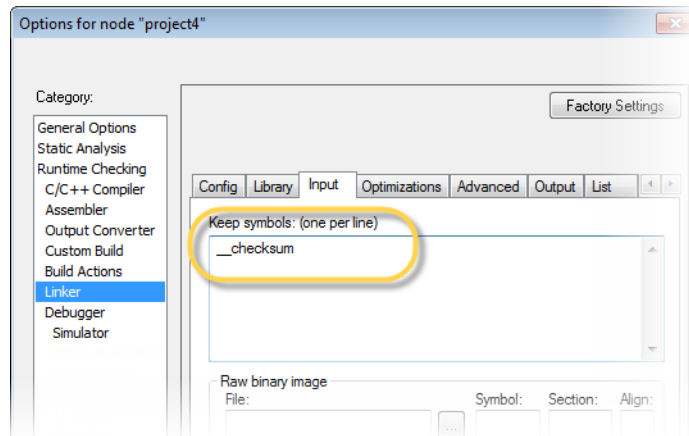
- By creating a global C/C++ or assembler constant symbol with a proper size, residing in a specific section—in this example, `.checksum`
- By using the linker option `--place_holder`.

For example, to allocate a 2-byte space for the symbol `__checksum` in the section `.checksum`, with alignment 4, specify:

```
--place_holder __checksum, 2, .checksum, 4
```

- 2 The `.checksum` section will only be included in your application if the section appears to be needed. If the checksum is not needed by the application itself, use the linker option `--keep=__checksum` (or the linker directive `keep`) to force the section to be included.

Alternatively, choose **Project>Options>Linker>Output** and specify `__checksum`:



- 3 To control the placement of the `.checksum` section, you must modify the linker configuration file. For example, it can look like this (note the handling of the block `CHECKSUM`):

```
define block CHECKSUM      { ro section .checksum };
place in ROM_region { ro, first block CHECKSUM };
```

Note: It is possible to skip this step, but in that case the `.checksum` section will automatically be placed with other read-only data.

- 4 When configuring `ielftool` 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, you can specify one memory range for which the checksum should be calculated. 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 209.



To run `ielftool` 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**, **Reverse byte order within word**, and **Checksum unit size**.



To run `ielftool` from the command line, specify the command, for example, like this:

```
ielftool --fill=0x00;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x8002-0x8FFF sourceFile.out
destinationFile.out
```

Note: `ielftool` needs an unstripped input ELF image. If you use the linker option `--strip`, remove it and use the `ielftool` option `--strip` instead.

The checksum will be created later on when you build your project and will be automatically placed in the specified symbol `__checksum` in the section `.checksum`.

5 You can specify several ranges instead of only one range.

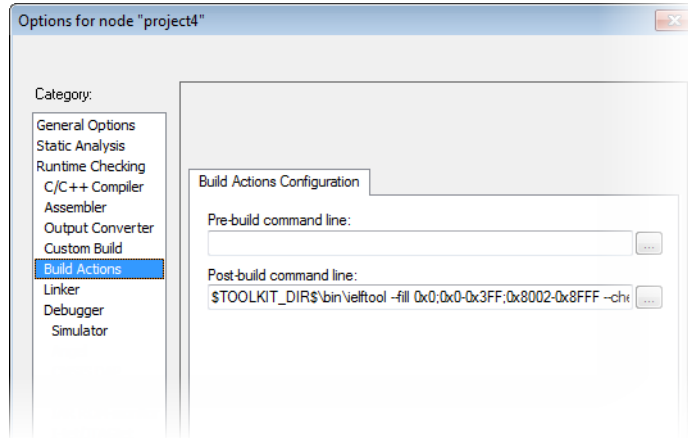


If you are using the IDE, perform these steps:

- Choose **Project>Options>Linker>Checksum** and make sure to deselect **Fill unused code memory**.
- Choose **Project>Options>Build Actions** and specify the ranges together with the rest of the required commands in the **Post-build command line** text field, for example like this:

```
$TOOLKIT_DIR$\bin\ielftool $PROJ_DIR$\debug\exe\output.out
$PROJ_DIR$\debug\exe\output.out
--fill 0x0;0x0-0x3FF;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x0-0x3FF;0x8002-0x8FFF
```

In your example, replace *output.out* with the name of your output file.



If you are using the command line, specify the ranges, for example like this:

```
ielftool output.out output.out --fill 0x0;0x0-0x3FF;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x0-0x3FF;0x8002-0x8FFF
```

In your example, replace *output.out* with the name of your output file.

- 6** 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 `ie1ftool`. 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 `r178\src\linker` directory of your product installation.

- 7** 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 `ie1ftool` 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 */
}

```

8 Build your application project and download it.

During the build, `ielftool` creates a checksum and places it in the specified symbol `__checksum` in the section `.checksum`.

9 Choose **Download and Debug** to start the C-SPY debugger.

During execution, the checksum calculated by `ielftool` 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, `ie1ftool` lists the ranges for which the checksum is calculated on `stdout` 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.
Compare the checksum placement with the checksum range and make sure they do not overlap. You can find information in the **Build** message window after `ie1ftool` has generated a checksum.
- Because pointers declared `__near` implicitly have some of the most significant bits set, make sure that the checksum calculated by your application uses a `__far` pointer instead of a `__near` pointer.
- 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, four zeros for a 4-byte checksum, and eight zeros for an 8-byte checksum.
- Carefully consider the number of bits in pointers when crossing 64-Kbyte boundaries.
For example, if the pointer `0xFFFF` is incremented by one, this typically results in the pointer `0x0` (and not `0x10000`) if the pointer is 16 bits.
- 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 `ie1ftool`. 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 RL78*.
- By default, a symbol that you have allocated in memory by using the linker option `--place_holder` is considered by C-SPY to be of the type `int`. If the size of the checksum is different than the size of an `int`, you can change the display format of the checksum symbol to match its size.

In the C-SPY **Watch** window, select the symbol and choose **Show As** from the context menu. Choose the display format that matches the size of the checksum symbol.

Patching symbol definitions using \$\$Super\$\$ and \$\$Sub\$\$

Using the \$\$Sub\$\$ and \$\$Super\$\$ special patterns, you can patch existing symbol definitions in situations where you would otherwise not be able to modify the symbol, for example, when a symbol is located in an external library or in ROM code.

The \$\$Super\$\$ special pattern identifies the original unpatched function used for calling the original function directly.

The \$\$Sub\$\$ special pattern identifies the new function that is called instead of the original function. You can use the \$\$Sub\$\$ special pattern to add processing before or after the original function.

AN EXAMPLE USING THE \$\$SUPER\$\$ AND \$\$SUB\$\$ PATTERNS

The following example shows how to use the \$\$Super\$\$ and \$\$Sub\$\$ patterns to insert a call to the function `ExtraFunc()` before the call to the legacy function `foo()`.

```
extern void ExtraFunc(void);
extern void $$Super$$foo(void);

/* this function is called instead of the original foo() */
void $$Sub$$foo(void)
{
    ExtraFunc(); /* does some extra setup work */
    $$Super$$foo(); /* calls the original foo() function */
    /* To avoid calling the original foo() function
       * omit the $$Super$$foo(); function call.
       */
}
}
```

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 and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- 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.
- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array.
- Using floating-point types on a microprocessor without a math co-processor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer parameter to point to `const` data might open for better optimizations in the calling function.

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 inefficient, both in terms of code size and execution speed. Therefore, 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.

For more information about floating-point types, see *Basic data types—floating-point types*, page 326.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The RL78 microcontroller requires that data in memory must be aligned. Each element in a structure must be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment at 2.

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 321.

Use the `#pragma pack` directive or the `__packed` data type attribute for a tighter layout of the structure. The drawback is that each access to an unaligned element in the structure will use more code.

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 371.

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 RL78 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 267, 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 __sfr
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char way: 1;
        unsigned char out: 1;
    };
} @ 0xFFF80;

/* The variables are used here. */
void Test(void)
{
    IOPORT = 0;
    way = 1;
    out = 1;
}
```

This declares an I/O register byte `IOPORT` at address `0xFFF80`. 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:

- **Code models**
By selecting a code model, you can control the default memory placement of functions. For more information, see *Code models and memory attributes for function storage*, page 77.
- **Data models**
By selecting a data model, you can control the default memory placement of variables and constants. For more information, see *Data models*, page 72.
- **Data memory attributes**
Using IAR-specific keywords or pragma directives, you can override the default placement of functions, variables, and constants. For more information, see *Using function memory attributes*, page 78 and *Using data memory attributes*, page 67, respectively.
- **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. Note that it is not possible to use this notation for absolute placement of individual functions. For more information, see *Data placement at an absolute location*, page 223.
- **The @ operator and the #pragma location directive for section placement.**
Using the @ operator or the #pragma location directive, you can place individual functions, variables, and constants in named sections. The placement of these sections can then be controlled by linker directives. For more information, see *Data and function placement in sections*, page 224.

- The `--code_section` option

Use the `--code_section` option to place functions in named sections, which is useful, for example, if you want to direct them to different fast or slow memories. To read more about the `--code_section` option, see *--code_section*, page 258.

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.

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 `__no_init` 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.

Other variables placed at an absolute address use the normal distinction between declaration and definition. For these variables, you must provide the definition in only one module, normally with an initializer. Other modules can refer to the variable by using an `extern` declaration, with or without an explicit address.

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 @ 0xF2000; /* 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. (The first case is useful for configuration parameters, because they are accessible from an external interface.) Both objects are placed in ROM. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0x12002
__no_init __far const int beta;           /* OK */

__far const int gamma @ 0x12004 = 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.

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 __far 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 __far 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 SECTIONS

The following methods can be used for placing data or functions in named sections other than default:

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named sections. The named section can either be a predefined section, or a user-defined section.
- The `--code_section` option can be used for placing functions, which are parts of the whole compilation unit, in named sections.

C++ static member variables can be placed in named sections just like any other static variable.

If you use your own sections, in addition to the predefined sections, the sections must also be defined in the linker configuration file.

Note: Take care when explicitly placing a variable or function in a predefined section 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 sections can be controlled from the linker configuration file.

For more information about sections, see the chapter *Section reference*.

Examples of placing variables in named sections

In the following examples, a data object is placed in a user-defined section. 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 section 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 linker will normally arrange for the correct type of initialization for each variable. If you want to control or suppress automatic initialization, you can use the `initialize` and `do not initialize` directives in the linker configuration file.

As usual, you can use memory attributes to select a memory for the variable. Note that you must as always ensure that the section is placed in the appropriate memory area when linking.

```
__near __no_init int alpha @ "MY_NEAR_NOINIT"; /* Placed in
                                                near*/
```

Examples of placing functions in named sections

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

void g(void) @ "MY_FUNCTIONS"
{
}

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

Specify a memory attribute to direct the function to a specific memory, and then modify the segment placement in the linker configuration file accordingly:

```
__near_func void f(void) @ "MY_NEAR_FUNC_FUNCTIONS";
```

Controlling compiler optimizations

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

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

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute 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 370, 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 272.

Note: Only one object file is generated, and therefore all symbols will be part of that object file.

If the whole application is compiled as one compilation unit, it is 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 264.

OPTIMIZATION LEVELS

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

Optimization level	Description
None (Best debug support)	Variables live through their entire scope Dead code elimination Redundant label elimination Redundant branch elimination
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope
Medium	Same as above, and: Live-dead analysis and optimization Code hoisting Peephole optimization Register content analysis and optimization Common subexpression elimination Static clustering
High (Balanced)	Same as above, and: Cross jumping Instruction scheduling (when optimizing for speed or balanced) Cross call (when optimizing for size or balanced) Loop unrolling Function inlining Code motion Type-based alias analysis

Table 21: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information, see *Fine-tuning enabled transformations*, page 228.

A high level of optimization might result in increased compile time, and will also most likely 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, or even occasionally display an incorrect value. 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.

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 280 and `optimize`, page 370). 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.

Note: 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 272) 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_inlining`, alternatively its equivalent in the IDE **Function inlining**, or the `#pragma optimize` directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Static clustering
- Cross call
- Instruction scheduling.

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 275.

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.

For information about the related pragma directive, see `unroll`, page 377.



To disable loop unrolling, use the command line option `--no_unroll`, see `--no_unroll`, page 279.

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 83.



To disable function inlining, use the command line option `--no_inline`, see `--no_inline`, page 276.

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 274.

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 277.

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. Therefore, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. If you use explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Static clustering

When static clustering is enabled, static and global variables that are defined within the same module are arranged so that variables that are accessed in the same function are

stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses.

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



For more information about the command line option, see `--no_clustering`, page 274.

Cross call

Common code sequences are extracted to local subroutines. This optimization, which is performed at optimization level High, can reduce code size, sometimes dramatically, on behalf of execution time and stack size. The resulting code might however be difficult to debug. This optimization cannot be disabled using the `#pragma optimize` directive.

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



For more information about related command line options, see `--no_cross_call`, page 275.

Instruction scheduling

The compiler features an instruction scheduler to increase the performance of the generated code. To achieve that goal, the scheduler rearranges the instructions to minimize the number of pipeline stalls emanating from resource conflicts within the microprocessor. Note that not all cores benefit from scheduling. The resulting code might be difficult to debug.

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



For more information about the command line option, see `--no_scheduling`, page 276.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code:

- *Writing optimization-friendly source code*, page 232
- *Saving stack space and RAM memory*, page 232
- *Function prototypes*, page 232
- *Integer types and bit negation*, page 233
- *Protecting simultaneously accessed variables*, page 234
- *Accessing special function registers*, page 234
- *Non-initialized variables*, page 235

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 therefore 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 229. 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 226.
- 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 159.

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.

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 in the range 0–255, which can never be equal to `0xFF7F`. Furthermore, it 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 343.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 332.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several RL78 devices are included in the IAR product installation. The header files are named `iodevice.h` and define the processor-specific special function registers (SFRs).

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

SFRs with bitfields are declared in the header file. This example is from `ior178.h`:

```
typedef struct
{
    unsigned char no0:1;
    unsigned char no1:1;
    unsigned char no2:1;
    unsigned char no3:1;
    unsigned char no4:1;
    unsigned char no5:1;
    unsigned char no6:1;
    unsigned char no7:1;
} __BITS8;

__sfr __no_init volatile union
{
    unsigned char PM0;
    __BITS8 PM0_bit;
} @ 0xFFF20;

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

void Test()
{
    /* Whole register access */
    PM0 = 0x12;

    /* Bitfield accesses */
    PM0_bit.no1 = 1;
    PM0_bit.no5 = 0;
}
```

You can also use the header files as templates when you create new header files for other RL78 devices. For information about the @ operator, see *Controlling data and function placement in memory*, page 222.

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 section, according to the specified memory keyword.

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.

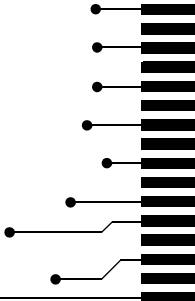
For more information, see `__no_init`, page 346.

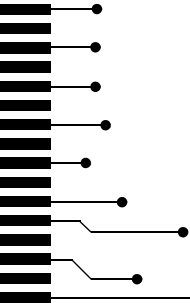
Note: To use this keyword, language extensions must be enabled, see `-e`, page 267. For more information, see `object_attribute`, page 369.

Part 2. Reference information

This part of the *IAR C/C++ Development Guide for RL78* contains these chapters:

- External interface details
- Compiler options
- Linker options
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- C/C++ standard library functions
- The linker configuration file
- Section reference
- The stack usage control file
- IAR utilities
- Implementation-defined behavior for Standard C++
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89.





External interface details

- Invocation syntax
- Include file search procedure
- Compiler output
- ILINK output
- Text encodings
- Reserved identifiers
- Diagnostics

Invocation syntax

You can use the compiler and linker either from the IDE or from the command line. See the *IDE Project Management and Building Guide for RL78* for information about using the build tools from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccr178 [options] [sourcefile] [options]
```

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

```
iccr178 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.

ILINK INVOCATION SYNTAX

The invocation syntax for ILINK is:

```
ilinkr178 [arguments]
```

Each argument is either a command-line option, an object file, or a library.

For example, when linking the object file `prog.o`, use this command:

```
ilinkr178 prog.o --config configfile
```

If no filename extension is specified for the linker configuration file, the configuration file must have the extension `icf`.

Generally, the order of arguments on the command line is not significant. There is, however, one exception: when you supply several libraries, the libraries are searched in the same order that they are specified on the command line. The default libraries are always searched last.

The output executable image will be placed in a file named `a.out`, unless the `-o` option is used.

If you run ILINK from the command line without any arguments, the ILINK 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 and to ILINK:

- Directly from the command line
 - Specify the options on the command line after the `iccr178` or `ilinkr178` commands, see *Invocation syntax*, page 239.
- Via environment variables
 - The compiler and linker automatically append the value of the environment variables to every command line, see *Environment variables*, page 241.
- Via a text file, using the `-f` option, see *-f*, page 268.

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
C_INCLUDE	Specifies directories to search for include files, for example: C_INCLUDE=c:\my_programs\embedded_workbench 8.n\r178\inc;c:\headers
QCCRL78	Specifies command line options, for example: QCCRL78=-lA asm.lst

Table 22: Compiler environment variables

This environment variable can be used with ILINK:

Environment variable	Description
ILINKRL78_CMD_LINE	Specifies command line options, for example: ILINKRL78_CMD_LINE=--config full.icf --silent

Table 23: ILINK environment variables

Include file search procedure

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

- The string found between the " " and `<>` in the `#include` directive is used verbatim as a source file name.
- 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 270.
 - 2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 241.
 - 3 The automatically set up library system include directories. See `--dlib_config`, page 265.
- 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:

```
iccrl78 ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

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

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

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

For more information, see *Overview of the preprocessor*, page 385.

Compiler output

The compiler can produce the following output:

- A linkable object file

The object files produced by the compiler use the industry-standard format ELF. By default, the object file has the filename extension `.o`.
- Optional list files

Various kinds of list files can be specified using the compiler option `-l`, see `-l`, page 270. 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. The file will have the filename extension `i`, by default.
- **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 246.
- **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 243.
- **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 and linker return 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 or linking 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 tool abort.
4	Internal errors occurred, making the tool abort.

Table 24: Error return codes

ILINK output

ILINK can produce the following output:

- An absolute executable image

The final output produced by the IAR ILINK Linker is an absolute object file containing the executable image that can be put into an EPROM, downloaded to a hardware emulator, or executed on your PC using the IAR C-SPY Debugger Simulator. By default, the file has the filename extension `out`. The output format is always in ELF, which optionally includes debug information in the DWARF format.

- Optional logging information

During operation, ILINK logs its decisions on `stdout`, and optionally to a file. For example, if a library is searched, whether a required symbol is found in a library module, or whether a module will be part of the output. Timing information for each ILINK subsystem is also logged.

- Optional map files

A linker map file—containing summaries of linkage, runtime attributes, memory, and placement, as well as an entry list—can be generated by the ILINK option `--map`, see `--map`, page 307. By default, the map file has the filename extension `map`.

- Diagnostic messages

Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in the optional map file. For more information about diagnostic messages, see *Diagnostics*, page 246.

- Error return codes

ILINK returns status information to the operating system which can be tested in a batch file, see *Error return codes*, page 243.

- Size information about used memory and amount of time

Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` and displayed on the screen.

Text encodings

Text files read or written by IAR tools can use a variety of text encodings:

- Raw

This is a backward-compatibility mode for C/C++ source files. Only 7-bit ASCII characters can be used in symbol names. Other characters can only be used in comments, literals, etc. This is the default source file encoding if there is no Byte Order Mark (BOM).

- The system default locale
The locale that you have configured your Windows OS to use.
- UTF-8
Unicode encoded as a sequence of 8-bit bytes, with or without a Byte Order Mark.
- UTF-16
Unicode encoded as a sequence of 16-bit words using a big-endian or little-endian representation. These files always start with a Byte Order Mark.

In any encoding other than Raw, you can use Unicode characters of the appropriate kind (alphabetic, numeric, etc) in the names of symbols.

When an IAR tool reads a text file with a Byte Order Mark, it will use the appropriate Unicode encoding, regardless of the any options set for input file encoding.

For source files without a Byte Order Mark, the compiler will use the Raw encoding, unless you specify the compiler option `--source_encoding`. See `--source_encoding`, page 285.

For other text input files, like the extended command line (`.xcl` files), without a Byte Order Mark, the IAR tools will use the system default locale unless you specify the compiler option `--utf8_text_in`, in which case UTF-8 will be used. See `--utf8_text_in`, page 288.

For compiler list files and preprocessor output, the same encoding as the main source file will be used by default. Other tools that generate text output will use the UTF-8 encoding by default. You can change this by using the compiler options `--text_out` and `--no_bom`. See `--text_out`, page 286 and `--no_bom`, page 273.

CHARACTERS AND STRING LITERALS

When you compile source code, characters (`x`) and string literals (`xx`) are handled as follows:

<code>'x', "xx"</code>	Characters in untyped character and string literals are copied verbatim, using the same encoding as in the source file.
<code>u8 "xx"</code>	Characters in UTF-8 string literals are converted to UTF-8.
<code>u'x', u"xx"</code>	Characters in UTF-16 character and string literals are converted to UTF-16.
<code>U'x', U"xx"</code>	Characters in UTF-32 character and string literals are converted to UTF-32.
<code>L'x', L"xx"</code>	Characters in wide character and string literals are converted to UTF-32.

Reserved identifiers

Some identifiers are reserved for use by the implementation. Some of the more important identifiers that the C/C++ standards reserve for any use are:

- Identifiers that contain a double underscore (__)
- Identifiers that begin with an underscore followed by an uppercase letter

In addition to this, the IAR tools reserve for any use:

- Identifiers that contain a double dollar sign (\$\$)
- Identifiers that contain a question mark (?)

More specific reservations are in effect in particular circumstances, see the C/C++ standards for more information.

Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

MESSAGE FORMAT FOR THE COMPILER

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.

MESSAGE FORMAT FOR THE LINKER

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from ILINK is produced in the form:

```
level[tag]: message
```

with these elements:

<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 and printed in the optional map file.

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler or linker finds a 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 284.

Warning

A diagnostic message that is produced when the compiler or linker finds a potential problem which is of concern, but which does not prevent completion of the compilation or linking. Warnings can be disabled by use of the command line option `--no_warnings`, see `--no_warnings`, page 279.

Error

A diagnostic message that is produced when the compiler or linker finds a serious error. An error will produce a non-zero exit code.

Fatal error

A diagnostic message produced when the compiler finds a condition that not only prevents code generation, but also makes further processing 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.

For information about the compiler options that are available for setting severity levels, see the chapter *Compiler options*.

For information about the pragma directives that are available for setting severity levels for the compiler, see the chapter *Pragma directives*.

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 or linker. 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 or of ILINK, which can be seen in the header of the list or map files generated by the compiler or by ILINK, respectively
- Your license number
- The exact internal error message text
- The files involved 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 240.

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\`:

```
iccr178 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:

```
iccrl78 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:
- ```
iccrl78 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:

```
iccrl78 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`:

```
iccrl78 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>--calling_convention</code> | Specifies the calling convention     |
| <code>--char_is_signed</code>     | Treats <code>char</code> as signed   |
| <code>--char_is_unsigned</code>   | Treats <code>char</code> as unsigned |
| <code>--code_model</code>         | Specifies the code model             |

Table 25: Compiler options summary

| Command line option                                   | Description                                                                                                 |
|-------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>--code_section</code>                           | Changes the default section for functions                                                                   |
| <code>--core</code>                                   | Specifies a CPU core                                                                                        |
| <code>--c++</code>                                    | Specifies Standard C++                                                                                      |
| <code>-D</code>                                       | Defines preprocessor symbols                                                                                |
| <code>--data_model</code>                             | Specifies the data model                                                                                    |
| <code>--debug</code>                                  | Generates debug information                                                                                 |
| <code>--dependencies</code>                           | Lists file dependencies                                                                                     |
| <code>--deprecated_feature_warnings</code>            | Enables/disables warnings for deprecated features                                                           |
| <code>--diag_error</code>                             | Treats these as errors                                                                                      |
| <code>--diag_remark</code>                            | Treats these as remarks                                                                                     |
| <code>--diag_suppress</code>                          | Suppresses these diagnostics                                                                                |
| <code>--diag_warning</code>                           | Treats these as warnings                                                                                    |
| <code>--diagnostics_tables</code>                     | Lists all diagnostic messages                                                                               |
| <code>--disable_div_mod_instructions</code>           | Disables the generation of DIVH/DIVHU instructions for the RL78 S3 core                                     |
| <code>--discard_unused_publics</code>                 | Discards unused public symbols                                                                              |
| <code>--dlib_config</code>                            | Uses the system include files for the DLIB library and determines which configuration of the library to use |
| <code>--do_explicit_zero_opt_in_named_sections</code> | For user-named sections, treats explicit initializations to zero as zero initializations                    |
| <code>--double</code>                                 | Forces the compiler to use 32-bit or 64-bit doubles                                                         |
| <code>-e</code>                                       | Enables language extensions                                                                                 |
| <code>--enable_restrict</code>                        | Enables the Standard C keyword <code>restrict</code>                                                        |
| <code>--error_limit</code>                            | Specifies the allowed number of errors before compilation stops                                             |
| <code>-f</code>                                       | Extends the command line                                                                                    |
| <code>--f</code>                                      | Extends the command line, optionally with a dependency.                                                     |
| <code>--generate_callt_runtime_library_calls</code>   | Generates <code>__callt</code> runtime-library calls                                                        |

Table 25: Compiler options summary (Continued)

| Command line option                               | Description                                                                                                                      |
|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>--generate_far_runtime_library_calls</code> | Generates <code>__far</code> runtime library calls                                                                               |
| <code>--guard_calls</code>                        | Enables guards for function static variable initialization                                                                       |
| <code>--header_context</code>                     | Lists all referred source files and header files                                                                                 |
| <code>-I</code>                                   | Specifies include file path                                                                                                      |
| <code>-l</code>                                   | Creates a list file                                                                                                              |
| <code>--macro_positions_in_diagnostics</code>     | Obtains positions inside macros in diagnostic messages                                                                           |
| <code>--max_cost_constexpr_call</code>            | Specifies the limit for <code>constexpr</code> evaluation cost                                                                   |
| <code>--max_depth_constexpr_call</code>           | Specifies the limit for <code>constexpr</code> recursion depth                                                                   |
| <code>--mfc</code>                                | Enables multi-file compilation                                                                                                   |
| <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>--near_const_location</code>                | Specifies near constants to be writable or mirrored from ROM                                                                     |
| <code>--no_bom</code>                             | Omits the Byte Order Mark for UTF-8 output files                                                                                 |
| <code>--no_call_frame_info</code>                 | Disables output of call frame information                                                                                        |
| <code>--no_clustering</code>                      | Disables static clustering optimizations                                                                                         |
| <code>--no_code_motion</code>                     | Disables code motion optimization                                                                                                |
| <code>--no_cross_call</code>                      | Disables cross-call optimization                                                                                                 |
| <code>--no_cse</code>                             | Disables common subexpression elimination                                                                                        |
| <code>--no_exceptions</code>                      | This option has no effect and is included for portability reasons                                                                |
| <code>--no_fragments</code>                       | Disables section fragment handling                                                                                               |

Table 25: Compiler options summary (Continued)

| Command line option                        | Description                                                                                                                     |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <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_rtti</code>                     | This option has no effect and is included for portability reasons                                                               |
| <code>--no_scheduling</code>               | Disables the instruction scheduler                                                                                              |
| <code>--no_size_constraints</code>         | Relaxes the normal restrictions for code size expansion when optimizing for speed.                                              |
| <code>--no_static_destruction</code>       | Disables destruction of C++ static variables at program exit                                                                    |
| <code>--no_system_include</code>           | Disables the automatic search for system include files                                                                          |
| <code>--no_tbaa</code>                     | Disables type-based alias analysis                                                                                              |
| <code>--no_typedefs_in_diagnostics</code>  | Disables the use of typedef names in diagnostics                                                                                |
| <code>--no_uniform_attribute_syntax</code> | Specifies the default syntax rules for IAR type attributes                                                                      |
| <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>--nonportable_path_warnings</code>   | Generates a warning when the path used for opening a source header file is not in the same case as the path in the file system. |
| <code>-O</code>                            | Sets the optimization level                                                                                                     |
| <code>-o</code>                            | Sets the object filename. Alias for <code>--output</code> .                                                                     |
| <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                                                                                                   |

Table 25: Compiler options summary (Continued)

| Command line option                          | Description                                                                                                 |
|----------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>--public_equ</code>                    | Defines a global named assembler label                                                                      |
| <code>-r</code>                              | Generates debug information. Alias for <code>--debug</code> .                                               |
| <code>--relaxed_fp</code>                    | Relaxes the rules for optimizing floating-point expressions                                                 |
| <code>--remarks</code>                       | Enables remarks                                                                                             |
| <code>--require_prototypes</code>            | Verifies that functions are declared before they are defined                                                |
| <code>--silent</code>                        | Sets silent operation                                                                                       |
| <code>--source_encoding</code>               | Specifies the encoding for source files                                                                     |
| <code>--stack_protection</code>              | Enables stack protection                                                                                    |
| <code>--strict</code>                        | Checks for strict compliance with Standard C/C++                                                            |
| <code>--system_include_dir</code>            | Specifies the path for system include files                                                                 |
| <code>--text_out</code>                      | Specifies the encoding for text output files                                                                |
| <code>--uniform_attribute_syntax</code>      | Specifies the same syntax rules for IAR type attributes as for <code>const</code> and <code>volatile</code> |
| <code>--use_c++_inline</code>                | Uses C++ inline semantics in C99                                                                            |
| <code>--use_paths_as_written</code>          | Use paths as written in debug information                                                                   |
| <code>--use_unix_directory_separators</code> | Uses / as directory separator in paths                                                                      |
| <code>--utf8_text_in</code>                  | Uses the UTF-8 encoding for text input files                                                                |
| <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                                                                              |
| <code>--workseg_area</code>                  | Enables a short address work section area with an optional size                                             |

Table 25: Compiler options summary (Continued)

## Descriptions of compiler options

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



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.<br><b>Note:</b> This option is mandatory when the MISRA C checking is enabled. |
| See also    | <i>C language overview</i> , page 183.                                                                                                            |



**Project>Options>C/C++ Compiler>Language 1>C dialect>C89**

### --calling\_convention

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |    |                                    |              |                                    |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|------------------------------------|--------------|------------------------------------|
| Syntax       | --calling_convention= <i>convention</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |    |                                    |              |                                    |
| Parameters   | <i>convention</i> is one of:<br><br><table> <tr> <td>v1</td> <td>Selects the V1 calling convention.</td> </tr> <tr> <td>v2 (default)</td> <td>Selects the V2 calling convention.</td> </tr> </table>                                                                                                                                                                                                                                                                                                                                                           | v1 | Selects the V1 calling convention. | v2 (default) | Selects the V2 calling convention. |
| v1           | Selects the V1 calling convention.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |    |                                    |              |                                    |
| v2 (default) | Selects the V2 calling convention.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |    |                                    |              |                                    |
| Description  | Use this option to specify the default calling convention for a module. All runtime modules in an application must use the same calling convention. However, note that it is possible to override this for individual functions, by using keywords.<br><br><b>Note:</b> If you are migrating from version 1.x of the compiler, note that the stack is now cleaned by the calling function, not by the called function, regardless of which calling convention you use. Assembler functions that take stack parameters must be updated before you can use them. |    |                                    |              |                                    |
| See also     | <i>Calling convention</i> , page 166.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |    |                                    |              |                                    |



**Project>Options>General Options>Target>Calling convention**



## --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**

## --code\_model

Syntax `--code_model={near|n|far|f}`

Parameters

|                             |                                                     |
|-----------------------------|-----------------------------------------------------|
| <code>near</code> (default) | Function calls reach the first 64 Kbytes of memory. |
| <code>far</code>            | Function calls reach the entire 1 Mbyte memory.     |

Description Use this option to select the code model. If you do not select a code model, the compiler uses the default code model. Note that all modules of your application must use the same code model.

See also *Code models and memory attributes for function storage*, page 77.



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

## --code\_section

Syntax

`--code_section=section_name`

Description

By default, the compiler places functions into a code section determined by the code model. Use the `--code_section` option to by default place all functions in a section with the name `section_name`. You can then assign this section to a fixed address in the linker configuration file. (The section name is case-sensitive.)

This is useful if you want to place your code in a different address range and you find the `@` notation, alternatively the `#pragma location` directive, insufficient. Note that any changes to the section names require corresponding modifications in the linker configuration file.

**Note:** This option only applies to functions that are placed in default memory, either explicitly or implicitly. If you use a function memory attribute that explicitly places the function in another section than the default section, that function will not be affected by the `--code_section` option.

See also

*Controlling data and function placement in memory*, page 222.



**Project>Options>C/C++ Compiler>Output>Code section**

## --core

Syntax

`--core={s1 | s2 | s3}`

Parameters

|              |                                                                                                |
|--------------|------------------------------------------------------------------------------------------------|
| s1           | Generates code for S1, the RL78 core with only one register bank and a multiplexed 8-bit bus.  |
| s2           | Generates code for S2, the core without instructions to support a hardware multiplier/divider. |
| s3 (default) | Generates code for S3, the core with instructions to support a hardware multiplier/divider.    |

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, the compiler generates code for the S3 core as default. Note that all modules of your application must use the same core.

The compiler supports all RL78 microcontroller cores and derivatives based on these cores.

**Project>Options>General Options>Target>Core****--c++**

|             |                                                                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--c++</code>                                                                                                                                       |
| Description | By default, the language supported by the compiler is C. If you use Standard C++, you must use this option to set the language the compiler uses to C++. |
| See also    | <i>Using C++</i> , page 191.                                                                                                                             |

**Project>Options>C/C++ Compiler>Language 1>C++**

and

**Project>Options>C/C++ Compiler>Language 1>C++ dialect>C++****-D**

|               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                     |              |                                      |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------------------------------------|--------------|--------------------------------------|
| Syntax        | <code>-D <i>symbol</i> [= <i>value</i>]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |               |                                     |              |                                      |
| Parameters    | <table> <tr> <td><i>symbol</i></td> <td>The name of the preprocessor symbol</td> </tr> <tr> <td><i>value</i></td> <td>The value of the preprocessor symbol</td> </tr> </table>                                                                                                                                                                                                                                                                                                                                   | <i>symbol</i> | The name of the preprocessor symbol | <i>value</i> | The value of the preprocessor symbol |
| <i>symbol</i> | The name of the preprocessor symbol                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |               |                                     |              |                                      |
| <i>value</i>  | The value of the preprocessor symbol                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |               |                                     |              |                                      |
| Description   | <p>Use this option to define a preprocessor symbol. If no value is specified, 1 is used. This option can be used one or more times on the command line.</p> <p>The option <code>-D</code> has the same effect as a <code>#define</code> statement at the top of the source file:</p> <pre>-D<i>symbol</i></pre> <p>is equivalent to:</p> <pre>#define <i>symbol</i> 1</pre> <p>To get the equivalence of:</p> <pre>#define FOO</pre> <p>specify the = sign but nothing after, for example:</p> <pre>-DFOO=</pre> |               |                                     |              |                                      |

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

## --data\_model

|             |                                                                                                                                                                                                                                                 |                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| Syntax      | <code>--data_model={near n far f huge h}</code>                                                                                                                                                                                                 |                                                                                                   |
| Parameters  | <code>near</code> (default)                                                                                                                                                                                                                     | Data is by default placed in the highest 64 Kbytes of memory.                                     |
|             | <code>far</code>                                                                                                                                                                                                                                | Data is by default placed in the entire 1 Mbyte of memory, but with a restriction on object size. |
|             | <code>huge</code>                                                                                                                                                                                                                               | Data is by default placed in the entire 1 Mbyte of memory.                                        |
| Description | Use this option to select the data model, which means a default placement of data objects. If you do not select a data model, the compiler uses the default data model. Note that all modules of your application must use the same data model. |                                                                                                   |
| See also    | <i>Data models</i> , page 72.                                                                                                                                                                                                                   |                                                                                                   |



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

## --debug, -r

|             |                                                                                                                                                                                         |  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--debug</code><br><code>-r</code>                                                                                                                                                 |  |
| Description | Use the <code>--debug</code> or <code>-r</code> option to make the compiler include information in the object modules required by the IAR C-SPY® Debugger and other symbolic debuggers. |  |
|             | <b>Note:</b> Including debug information will make the object files larger than otherwise.                                                                                              |  |



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

## --dependencies

|            |                                                               |                                          |
|------------|---------------------------------------------------------------|------------------------------------------|
| Syntax     | <code>--dependencies=[i m 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)       |

- s Suppresses system files
- + Gives the same output as `-o`, but with the filename extension `d`

See also *Rules for specifying a filename or directory as parameters*, page 250.

**Description** Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension `i`.

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

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

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

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

An example of using `--dependencies` with a popular make utility, such as `gmake` (GNU make):

- 1 Set up the rule for compiling files to be something like:

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

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

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

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

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



This option is not available in the IDE.

## --deprecated\_feature\_warnings

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--deprecated_feature_warnings=[+ -] feature[, [+ -] feature, ...]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                           |
| Parameters  | <i>feature</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | A feature can be <code>attribute_syntax</code> , <code>preprocessor_extensions</code> , or <code>segment_pragmas</code> . |
| Description | <p>Use this option to disable or enable warnings for the use of a deprecated feature. The deprecated features are:</p> <ul style="list-style-type: none"> <li>• <code>attribute_syntax</code><br/>See <code>--uniform_attribute_syntax</code>, page 287, <code>--no_uniform_attribute_syntax</code>, page 278, and <i>Syntax for type attributes used on data objects</i>, page 336.</li> <li>• <code>preprocessor_extensions</code></li> <li>• <code>segment_pragmas</code><br/>See the pragma directives <code>dataseg</code>, <code>constseg</code>, and <code>memory</code>. Use the <code>#pragma location</code> and <code>#pragma default_variable_attributes</code> directives instead.</li> </ul> <p>Because the deprecated features will be removed in a future version of the IAR C/C++ compiler, it is prudent to remove the use of them in your source code. To do this, enable warnings for a deprecated feature. For each warning, rewrite your code so that the deprecated feature is no longer used.</p> |                                                                                                                           |



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

## --diag\_error

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



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

## --diag\_remark

|             |                                                                                                                                                                                                                                                                                           |                                                                                        |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_remark=tag[, tag, ...]</code>                                                                                                                                                                                                                                                |                                                                                        |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                                | The number of a diagnostic message, for example, the message number <code>Pe177</code> |
| 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. |                                                                                        |
|             | <b>Note:</b> By default, remarks are not displayed—use the <code>--remarks</code> option to display them.                                                                                                                                                                                 |                                                                                        |



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

## --diag\_suppress

|             |                                                                                                                                                            |                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_suppress=tag[, tag, ...]</code>                                                                                                               |                                                                                        |
| Parameters  | <i>tag</i>                                                                                                                                                 | The number of a diagnostic message, for example, the message number <code>Pe117</code> |
| Description | Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line. |                                                                                        |



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

## --diag\_warning

|            |                                             |                                                                                        |
|------------|---------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax     | <code>--diag_warning=tag[, tag, ...]</code> |                                                                                        |
| Parameters | <i>tag</i>                                  | The number of a diagnostic message, for example, the message number <code>Pe826</code> |

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



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

## --diagnostics\_tables

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

**Parameters** See *Rules for specifying a filename or directory as parameters*, page 250.

**Description** Use this option to list all possible diagnostic messages to 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.

Typically, this option cannot be given together with other options.



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

## --disable\_div\_mod\_instructions

**Syntax** `--disable_div_mod_instructions`

**Description** Use this option to disable the generation of DIVH/DIVHU instructions for the S3 core.



**Project>Options>C/C++ Compiler>Optimizations>Disable DIV/MOD instructions.**

## --discard\_unused\_publics

**Syntax** `--discard_unused_publics`

**Description** Use this option to discard unused public functions and variables when compiling with the `--mfc` compiler option.

**Note:** Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output. Use the object attribute `__root` to keep symbols that are used from outside the compilation unit, for example, interrupt handlers.



If the symbol does not have the `__root` attribute and is defined in the library, the library definition will be used instead.

See also

`--mfc`, page 272 and *Multi-file compilation units*, page 226.



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

## --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 250.                                                                                                                                                                                                                                            |
| <i>config</i>   | The default configuration file for the specified configuration will be used. Choose between: <ul style="list-style-type: none"> <li><code>none</code>, no configuration will be used</li> <li><code>normal</code>, the normal library configuration will be used (default)</li> <li><code>full</code>, the full library configuration will be used.</li> </ul> |

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.

You can find the library object files and the library configuration files in the directory `r178\lib`. For examples and information about prebuilt runtime libraries, see *Prebuilt runtime libraries*, page 132.

If you build your own customized runtime library, you can also create a corresponding customized library configuration file to specify to the compiler. For more information, see *Customizing and building your own runtime library*, page 129.



To set related options, choose:

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

## --do\_explicit\_zero\_opt\_in\_named\_sections

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--do_explicit_zero_opt_in_named_sections</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Description | <p>By default, the compiler treats static initialization of variables explicitly and implicitly initialized to zero the same, except for variables which are to be placed in user-named sections. For these variables, an explicit zero initialization is treated as a copy initialization, that is the same way as variables statically initialized to something other than zero.</p> <p>Use this option to disable the exception for variables in user-named sections, and thus treat explicit initializations to zero as zero initializations, not copy initializations.</p> |
| Example     | <pre>int var1;                // Implicit zero init -&gt; zero initied int var2 = 0;            // Explicit zero init -&gt; zero initied int var3 = 7;            // Not zero init -&gt; copy initied int var4 @ "MYDATA";    // Implicit zero init -&gt; zero initied int var5 @ "MYDATA" = 0; // Explicit zero init -&gt; copy initied                         // If option specified, then zero initied int var6 @ "MYDATA" = 7; // Not zero init -&gt; copy initied</pre>                                                                                                   |



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

## --double

|              |                                                                                                                                                                                                                                                                         |              |                         |    |                         |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|-------------------------|----|-------------------------|
| Syntax       | <code>--double={32 64}</code>                                                                                                                                                                                                                                           |              |                         |    |                         |
| Parameters   | <table> <tr> <td>32 (default)</td> <td>32-bit doubles are used</td> </tr> <tr> <td>64</td> <td>64-bit doubles are used</td> </tr> </table>                                                                                                                              | 32 (default) | 32-bit doubles are used | 64 | 64-bit doubles are used |
| 32 (default) | 32-bit doubles are used                                                                                                                                                                                                                                                 |              |                         |    |                         |
| 64           | 64-bit doubles are used                                                                                                                                                                                                                                                 |              |                         |    |                         |
| Description  | <p>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.</p> |              |                         |    |                         |
| See also     | <i>Basic data types—floating-point types</i> , page 326.                                                                                                                                                                                                                |              |                         |    |                         |



**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.<br><br><b>Note:</b> The <code>-e</code> option and the <code>--strict</code> option cannot be used at the same time. |
| See also    | <i>Enabling language extensions, page 185.</i>                                                                                                                                                                                                                                                                                                                         |



**Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions**

**Note:** By default, this option is selected in the IDE.

**--enable\_restrict**

|             |                                                                                                                                                                                                                                                                             |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--enable_restrict</code>                                                                                                                                                                                                                                              |
| Description | Enables the Standard C keyword <code>restrict</code> in C89 and C++. By default, <code>restrict</code> is recognized in Standard C and <code>__restrict</code> is always recognized.<br><br>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  | <code>n</code> The number of errors before the compiler stops the compilation. <code>n</code> must be a positive integer. 0 indicates no limit.                  |
| Description | Use the <code>--error_limit</code> option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed. |



This option is not available in the IDE.

## **-f**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>-f filename</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 250.                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description | <p>Use this option to make the compiler read command line options from the named file, with the default filename extension <code>.xcl</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.</p> <p>Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.</p> |
| See also    | <code>-f</code> , page 268.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |



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

## **--f**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--f filename</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 250.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Description | <p>Use this option to make the compiler read command line options from the named file, with the default filename extension <code>.xcl</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.</p> <p>Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.</p> <p>If you use the compiler option <code>--dependencies</code>, extended command line files specified using <code>--f</code> will generate a dependency, but those specified using <code>-f</code> will not generate a dependency.</p> |
| See also    | <code>--dependencies</code> , page 260 and <code>-f</code> , page 268.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |



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

## --generate\_callt\_runtime\_library\_calls

Syntax `--generate_callt_runtime_library_calls`

Description Use this option to generate `__callt` runtime library calls, which means the `CALLT` instruction is used when calling the most frequently used runtime library routines. This will reduce code size.



**Project>Options>C/C++ Compiler>Optimizations>Enable callt runtime library calls.**

## --generate\_far\_runtime\_library\_calls

Syntax `--generate_far_runtime_library_calls`

Description Use this option to generate `__far` runtime library calls to assembler support routines. This option can be used together with the corresponding assembler option to move the support routines from near to far memory when building a customized library. Note that the startup code will still be located in near memory.

This option sets the runtime model attribute `__far_rt_calls` to true.

**Note:** This option overrides the option

`--generate_callt_runtime_library_calls`.



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

## --guard\_calls

Syntax `--guard_calls`

Description Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.

See also *Managing a multithreaded environment*, page 157.



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

## --header\_context

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



This option is not available in the IDE.

## -I

|             |                                                                                                                             |
|-------------|-----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | -I <i>path</i>                                                                                                              |
| Parameters  | <i>path</i> The search path for #include files                                                                              |
| Description | Use this option to specify the search paths for #include files. This option can be used more than once on the command line. |
| See also    | <i>Include file search procedure</i> , page 241.                                                                            |



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

## -l

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                            |             |                     |   |                                                      |   |                                                                                                                                                                                                                                           |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|---------------------|---|------------------------------------------------------|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | -l [a A b B c C D] [N] [H] { <i>filename</i>   <i>directory</i> }                                                                                                                                                                                                                                                                                                                                                                          |             |                     |   |                                                      |   |                                                                                                                                                                                                                                           |
| 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 -la, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is 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 -la, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is 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 -la, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *                                                                                                                                                                                                  |             |                     |   |                                                      |   |                                                                                                                                                                                                                                           |

|             |                                                                                                                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| B           | Basic assembler list file. This file has the same contents as a list file produced with <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 250.

#### Description

Use this option to generate an assembler or C/C++ listing to a file.

**Note:** This option can be used one or more times on the command line.



To set related options, choose:

**Project>Options>C/C++ Compiler>List**

## --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**.

## **--max\_cost\_constexpr\_call**

|             |                                                                                                                                                                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--max_cost_constexpr_call=<i>limit</i></code>                                                                                                                                                                                                                                                  |
| Parameters  | <i>limit</i> The number of calls and loop iterations. The default is 2000000.                                                                                                                                                                                                                        |
| Description | Use this option to specify an upper limit for the <i>cost</i> for folding a top-level <code>constexpr</code> call (function or constructor). The cost is a combination of the number of calls interpreted and the number of loop iterations performed during the interpretation of a top-level call. |



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

## **--max\_depth\_constexpr\_call**

|             |                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--max_depth_constexpr_call=<i>limit</i></code>                                                                                     |
| Parameters  | <i>limit</i> The depth of recursion. The default is 1000.                                                                                |
| Description | Use this option to specify the maximum depth of recursion for folding a top-level <code>constexpr</code> call (function or constructor). |



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

## **--mfc**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--mfc</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Description | Use this option to enable <i>multi-file compilation</i> . This means that the compiler compiles one or several source files specified on the command line as one unit, which enhances interprocedural optimizations.<br><br><b>Note:</b> The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the <code>-o</code> compiler option and specify a certain output file. |
| Example     | <code>iccr178 myfile1.c myfile2.c myfile3.c --mfc</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |



See also

`--discard_unused_publics`, page 264, `--output, -o`, page 281, and *Multi-file compilation units*, page 226.



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

## `--near_const_location`

Syntax

`--near_const_location [RAM|ROM0|ROM1]`

Parameters

|      |                                                                                                                               |
|------|-------------------------------------------------------------------------------------------------------------------------------|
| RAM  | Constants are located in RAM, in the range 0xF0000–0xFFFFF                                                                    |
| ROM0 | Constants are located in ROM, in the range 0x00000–0x0FFFF, and are mirrored by hardware to RAM, in the range 0xF0000–0xFFFFF |
| ROM1 | Constants are located in ROM, in the range 0x10000–0x1FFFF, and are mirrored by hardware to RAM, in the range 0xF0000–0xFFFFF |

Description

Use this option to specify the location for `__near` declared constants and strings: in RAM or mirrored from ROM.

For a detailed description of how this works and of the available memory ranges, see the chip manufacturer's documentation.



**Project>Options>General Options>Target>Near constant location**

## `--no_bom`

Syntax

`--no_bom`

Description

Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

See also

`--text_out`, page 286, and *Text encodings*, page 244.



**Project>Options>C/C++ Compiler>Encodings>Text output file encoding**

## --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 177.



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

## --no\_clustering

Syntax `--no_clustering`

Description Use this option to disable static clustering optimizations.

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

See also *Static clustering*, page 230.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Static clustering**

## --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 229.



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

## --no\_cross\_call

Syntax `--no_cross_call`

Description Use this option to disable the cross-call optimization.

**Note:** This option has no effect at optimization levels below High, or when optimizing for Speed, because cross-call optimization is not enabled then.

See also *Cross call*, page 231.



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

## --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 229.



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

## --no\_fragments

Syntax `--no_fragments`

Description Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and further minimize the size of the executable image. When you use this option, this information is not output in the object files.

See also *Keeping symbols and sections*, page 109.



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

## --no\_inline

Syntax `--no_inline`

Description Use this option to disable function inlining.

See also *Inlining functions*, page 83.



**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 386.



This option is not available in the IDE.

## --no\_scheduling

Syntax `--no_scheduling`

Description Use this option to disable the instruction scheduler.

**Note:** This option has no effect at optimization levels below High, or when optimizing for Size, because instruction scheduling is not enabled then.

See also *Instruction scheduling*, page 231.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling**

## --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 228.



**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

*Setting up the atexit limit*, page 110.



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\_config*, page 265, and *--system\_include\_dir*, page 286.



**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 230.



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

## **--no\_typedefs\_in\_diagnostics**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_typedefs_in_diagnostics</code>                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Description | Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.                                                                                                                                                        |
| Example     | <pre>typedef int (*MyPtr)(char const *); MyPtr p = "My text string";</pre> <p>will give an error message like this:</p> <pre>Error[Pe144]: a value of type "char *" cannot be used to initialize an entity of type "MyPtr"</pre> <p>If the <code>--no_typedefs_in_diagnostics</code> option is used, the error message will be like this:</p> <pre>Error[Pe144]: a value of type "char *" cannot be used to initialize an entity of type "int (*)(char const *)"</pre> |



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

## **--no\_uniform\_attribute\_syntax**

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_uniform_attribute_syntax</code>                                                                          |
| Description | Use this option to apply the default syntax rules to IAR type attributes specified before a type specifier.         |
| See also    | <i>--uniform_attribute_syntax</i> , page 287 and <i>Syntax for type attributes used on data objects</i> , page 336. |



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 229.



**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.

## --nonportable\_path\_warnings

Syntax `--nonportable_path_warnings`

Description Use this option to make the compiler generate a warning when characters in the path used for opening a source file or header file are lower case instead of upper case, or vice versa, compared with the path in the file system.



This option is not available in the IDE.

## **-O**

Syntax `-O[n|l|m|h|hs|hz]`

### Parameters

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

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

### Description

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

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

### See also

*Controlling compiler optimizations*, page 226.



**Project>Options>C/C++ Compiler>Optimizations**

## **--only\_stdout**

Syntax `--only_stdout`

### Description

Use this option to make the compiler use the standard output stream (`stdout`), and messages that are normally directed to the error output stream (`stderr`).



This option is not available in the IDE.



## --output, -o

|             |                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--output {filename directory}</code><br><code>-o {filename directory}</code>                                                                                                                                                                             |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 250.                                                                                                                                                                              |
| Description | By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension <code>.o</code> . Use this option to explicitly specify a different output filename for the object code output. |



This option is not available in the IDE.

## --pending\_instantiations

|             |                                                                                                                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--pending_instantiations number</code>                                                                                                                                                                         |
| Parameters  | <i>number</i> 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 250.                                                                                                                                                                                                                                                                                                                                                                                  |
| Description | Use this option to list all symbols defined by the compiler or on the command line. (Symbols defined in the source code are not listed.) When using this option, make sure to also use the same options as for the rest of your project.<br><br>If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the <code>predef</code> filename extension. |

**Note:** 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 <i>includefile</i></code>                                                                                                                                                                                                                  |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 250.                                                                                                                                                                             |
| 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] [s]] {<i>filename</i>   <i>directory</i>}</code>                                                                                                                                                                                        |   |                  |   |                 |   |                           |
| Parameters  | <table> <tr> <td>c</td> <td>Include comments</td> </tr> <tr> <td>n</td> <td>Preprocess only</td> </tr> <tr> <td>s</td> <td>Suppress #line directives</td> </tr> </table> <p>See also <i>Rules for specifying a filename or directory as parameters</i>, page 250.</p> | c | Include comments | n | Preprocess only | s | Suppress #line directives |
| c           | Include comments                                                                                                                                                                                                                                                      |   |                  |   |                 |   |                           |
| n           | Preprocess only                                                                                                                                                                                                                                                       |   |                  |   |                 |   |                           |
| s           | Suppress #line directives                                                                                                                                                                                                                                             |   |                  |   |                 |   |                           |
| Description | Use this option to generate preprocessed output to a named file.                                                                                                                                                                                                      |   |                  |   |                 |   |                           |



**Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file**

## --public\_eq

|               |                                                                                                            |               |                                                |
|---------------|------------------------------------------------------------------------------------------------------------|---------------|------------------------------------------------|
| Syntax        | <code>--public_eq <i>symbol</i> [=value]</code>                                                            |               |                                                |
| Parameters    | <table> <tr> <td><i>symbol</i></td> <td>The name of the assembler symbol to be defined</td> </tr> </table> | <i>symbol</i> | The name of the assembler symbol to be defined |
| <i>symbol</i> | The name of the assembler symbol to be defined                                                             |               |                                                |

|             |                                                                                                                                                                                                                                |                                                   |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------|
|             | value                                                                                                                                                                                                                          | An optional value of the defined assembler symbol |
| Description | This option is equivalent to defining a label in assembler language using the <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.

## --relaxed\_fp

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--relaxed_fp</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description | <p>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:</p> <ul style="list-style-type: none"> <li>• The expression consists of both single and double-precision values</li> <li>• The double-precision values can be converted to single precision without loss of accuracy</li> <li>• The result of the expression is converted to single precision.</li> </ul> <p><b>Note:</b> Performing the calculation in single precision instead of double precision might cause a loss of accuracy.</p> |

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Example | <pre>float F(float a, float b) {     return a + b * 3.0; }</pre> <p>The C standard states that <code>3.0</code> in this example has the type <code>double</code> and therefore the whole expression should be evaluated in <code>double</code> precision. However, when the <code>--relaxed_fp</code> option is used, <code>3.0</code> will be converted to <code>float</code> and the whole expression can be evaluated in <code>float</code> precision.</p> |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



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 247.



**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**

## --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.

## --source\_encoding

|             |                                                                                                                                                                                                            |                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
| Syntax      | <code>--source_encoding {locale utf8}</code>                                                                                                                                                               |                                                            |
| Parameters  | <code>locale</code>                                                                                                                                                                                        | The default source encoding is the system locale encoding. |
|             | <code>utf8</code>                                                                                                                                                                                          | The default source encoding is the UTF-8 encoding.         |
| Description | When reading a source file with no Byte Order Mark (BOM), use this option to specify the encoding. If this option is not specified and the source file does not have a BOM, the Raw encoding will be used. |                                                            |
| See also    | <i>Text encodings</i> , page 244.                                                                                                                                                                          |                                                            |



**Project>Options>C/C++ Compiler>Encodings>Default source file encoding**

## --stack\_protection

|             |                                                                                              |  |
|-------------|----------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--stack_protection</code>                                                              |  |
| Description | Use this option to enable stack protection for the functions that are considered to need it. |  |
| See also    | <i>Stack protection</i> , page 85.                                                           |  |



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

## --strict

|             |                                                                                                                                                                                              |  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--strict</code>                                                                                                                                                                        |  |
| 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++. |  |
|             | <b>Note:</b> 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 185.                                                                                                                                              |  |



**Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict**

## **--system\_include\_dir**

|             |                                                                                                                                                                                                                                                             |                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--system_include_dir path</code>                                                                                                                                                                                                                      |                                                                                                                         |
| Parameters  | <code>path</code>                                                                                                                                                                                                                                           | The path to the system include files, see <i>Rules for specifying a filename or directory as parameters</i> , page 250. |
| Description | By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location. |                                                                                                                         |
| See also    | <code>--dlib_config</code> , page 265, and <code>--no_system_include</code> , page 277.                                                                                                                                                                     |                                                                                                                         |



This option is not available in the IDE.

## **--text\_out**

|             |                                                                                                                                                                                                                                                                                                                                                                 |                                        |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| Syntax      | <code>--text_out {utf8 utf16le utf16be locale}</code>                                                                                                                                                                                                                                                                                                           |                                        |
| Parameters  | <code>utf8</code>                                                                                                                                                                                                                                                                                                                                               | Uses the UTF-8 encoding                |
|             | <code>utf16le</code>                                                                                                                                                                                                                                                                                                                                            | Uses the UTF-16 little-endian encoding |
|             | <code>utf16be</code>                                                                                                                                                                                                                                                                                                                                            | Uses the UTF-16 big-endian encoding    |
|             | <code>locale</code>                                                                                                                                                                                                                                                                                                                                             | Uses the system locale encoding        |
| Description | Use this option to specify the encoding to be used when generating a text output file. The default for the compiler list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM). If you want text output in UTF-8 encoding without a BOM, use the option <code>--no_bom</code> . |                                        |
| See also    | <code>--no_bom</code> , page 273 and <i>Text encodings</i> , page 244.                                                                                                                                                                                                                                                                                          |                                        |



**Project>Options>C/C++ Compiler>Encodings>Text output file encoding**

## --uniform\_attribute\_syntax

|             |                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--uniform_attribute_syntax</code>                                                                                                                                                                                                                                                                                                                                                                                  |
| Description | By default, an IAR type attribute specified before the type specifier applies to the object or typedef itself, and not to the type specifier, as <code>const</code> and <code>volatile</code> do. If you specify this option, IAR type attributes obey the same syntax rules as <code>const</code> and <code>volatile</code> .<br><br>The default for IAR type attributes is to <i>not</i> use uniform attribute syntax. |
| See also    | <code>--no_uniform_attribute_syntax</code> , page 278 and <i>Syntax for type attributes used on data objects</i> , page 336.                                                                                                                                                                                                                                                                                             |



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

## --use\_c++\_inline

|             |                                                                                                                                                                 |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--use_c++_inline</code>                                                                                                                                   |
| Description | Standard C uses slightly different semantics for the <code>inline</code> keyword than C++ does. Use this option if you want C++ semantics when you are using C. |
| See also    | <i>Inlining functions</i> , page 83.                                                                                                                            |



**Project>Options>C/C++ Compiler>Language 1>C dialect>C++ inline semantics**

## --use\_paths\_as\_written

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--use_paths_as_written</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Description | By default, the compiler ensures that all paths in the debug information are absolute, even if not originally specified that way.<br><br>If you use this option, paths that were originally specified as relative will be relative in the debug information.<br><br>The paths affected by this option are: <ul style="list-style-type: none"> <li>• the paths to source files</li> <li>• the paths to header files that are found using an include path that was specified as relative</li> </ul> |



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

## **--use\_unix\_directory\_separators**

|             |                                                                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--use_unix_directory_separators</code>                                                                                                                                                                          |
| Description | Use this option to make DWARF debug information use / (instead of \) as directory separators in file paths.<br><br>This option can be useful if you have a debugger that requires directory separators in UNIX style. |



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

## **--utf8\_text\_in**

|             |                                                                                                                                                                                                     |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--utf8_text_in</code>                                                                                                                                                                         |
| Description | Use this option to specify that the compiler shall use UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).<br><br><b>Note:</b> This option does not apply to source files. |
| See also    | <i>Text encodings</i> , page 244.                                                                                                                                                                   |



**Project>Options>C/C++ Compiler>Encodings>Default input file encoding**

## **--version**

|             |                                                                                             |
|-------------|---------------------------------------------------------------------------------------------|
| Syntax      | <code>--version</code>                                                                      |
| 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 variable length arrays in C code. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the --c89 compiler option.<br><br><b>Note:</b> --vla should not be used together with the <code>longjmp</code> library function, as that can lead to memory leakages. |
| See also    | <i>C language overview</i> , page 183.                                                                                                                                                                                                                                                                                                              |



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 263.



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

## **--workseg\_area**

Syntax

`--workseg_area[=size]`

Parameters

*size*                      The size of the workseg area. If the parameter is omitted, the size is by default 20. The maximum value is 128.

Description

Specifies the space reserved in saddr memory for the `.wrkseg` section, which is used as the *register* variable work area.

**Note:** There is an overhead in saving and restoring the work area at each function entry and exit, unless the modifier `__no_save` is used with the function.

See also

*Short address working area*, page 74 and *.wrkseg*, page 447.



**Project>Options>C/C++ Compiler>Optimizations>Short address work area.**

# Linker options

- Summary of linker options
- Descriptions of linker options

For general syntax rules, see *Options syntax*, page 249.

---

## Summary of linker options

This table summarizes the linker options:

| Command line option                      | Description                                                                                      |
|------------------------------------------|--------------------------------------------------------------------------------------------------|
| <code>--call_graph</code>                | Produces a call graph file in XML format                                                         |
| <code>--config</code>                    | Specifies the linker configuration file to be used by the linker                                 |
| <code>--config_def</code>                | Defines symbols for the configuration file                                                       |
| <code>--config_search</code>             | Specifies more directories to search for linker configuration files                              |
| <code>--cpp_init_routine</code>          | Specifies a user-defined C++ dynamic initialization routine                                      |
| <code>--debug_lib</code>                 | Uses the C-SPY debug library                                                                     |
| <code>--default_to_complex_ranges</code> | Makes <code>complex_ranges</code> the default decompressor in <code>initialize</code> directives |
| <code>--define_symbol</code>             | Defines symbols that can be used by the application                                              |
| <code>--dependencies</code>              | Lists file dependencies                                                                          |
| <code>--diag_error</code>                | Treats these message tags as errors                                                              |
| <code>--diag_remark</code>               | Treats these message tags as remarks                                                             |
| <code>--diag_suppress</code>             | Suppresses these diagnostic messages                                                             |
| <code>--diag_warning</code>              | Treats these message tags as warnings                                                            |
| <code>--diagnostics_tables</code>        | Lists all diagnostic messages                                                                    |
| <code>--enable_stack_usage</code>        | Enables stack usage analysis                                                                     |
| <code>--entry</code>                     | Treats the symbol as a root symbol and as the start of the application                           |

*Table 26: Linker options summary*

| Command line option                          | Description                                                                                                                                                                            |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--entry_list_in_address_order</code>   | Generates an additional entry list in the map file sorted in address order                                                                                                             |
| <code>--error_limit</code>                   | Specifies the allowed number of errors before linking stops                                                                                                                            |
| <code>--export_built_in_config</code>        | Produces an <code>icf</code> file for the default configuration                                                                                                                        |
| <code>-f</code>                              | Extends the command line                                                                                                                                                               |
| <code>--f</code>                             | Extends the command line, optionally with a dependency.                                                                                                                                |
| <code>--force_output</code>                  | Produces an output file even if errors occurred                                                                                                                                        |
| <code>--image_input</code>                   | Puts an image file in a section                                                                                                                                                        |
| <code>--inline</code>                        | Inlines small routines                                                                                                                                                                 |
| <code>--keep</code>                          | Forces a symbol to be included in the application                                                                                                                                      |
| <code>-L</code>                              | Specifies more directories to search for object and library files. Alias for <code>--search</code> .                                                                                   |
| <code>--log</code>                           | Enables log output for selected topics                                                                                                                                                 |
| <code>--log_file</code>                      | Directs the log to a file                                                                                                                                                              |
| <code>--mangled_names_in_messages</code>     | Adds mangled names in messages                                                                                                                                                         |
| <code>--manual_dynamic_initialization</code> | Suppresses automatic initialization during system startup                                                                                                                              |
| <code>--map</code>                           | Produces a map file                                                                                                                                                                    |
| <code>--merge_duplicate_sections</code>      | Merges equivalent read-only sections                                                                                                                                                   |
| <code>--misrac1998</code>                    | Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .                                                                 |
| <code>--misrac2004</code>                    | Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .                                                                 |
| <code>--misrac_verbose</code>                | Enables verbose logging of MISRA C checking. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> and the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> . |
| <code>--no_bom</code>                        | Omits the Byte Order Mark from UTF-8 output files                                                                                                                                      |
| <code>--no_entry</code>                      | Sets the entry point to zero                                                                                                                                                           |
| <code>--no_fragments</code>                  | Disables section fragment handling                                                                                                                                                     |

Table 26: Linker options summary (Continued)

| Command line option                  | Description                                                                                                           |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>--no_free_heap</code>          | Uses the smallest possible heap implementation                                                                        |
| <code>--no_inline</code>             | Excludes functions from small function inlining                                                                       |
| <code>--no_library_search</code>     | Disables automatic runtime library search                                                                             |
| <code>--no_locals</code>             | Removes local symbols from the ELF executable image.                                                                  |
| <code>--no_range_reservations</code> | Disables range reservations for absolute symbols                                                                      |
| <code>--no_remove</code>             | Disables removal of unused sections                                                                                   |
| <code>--no_vfe</code>                | Disables Virtual Function Elimination                                                                                 |
| <code>--no_warnings</code>           | Disables generation of warnings                                                                                       |
| <code>--no_wrap_diagnostics</code>   | Does not wrap long lines in diagnostic messages                                                                       |
| <code>-o</code>                      | Sets the object filename. Alias for <code>--output</code> .                                                           |
| <code>--only_stdout</code>           | Uses standard output only                                                                                             |
| <code>--output</code>                | Sets the object filename                                                                                              |
| <code>--place_holder</code>          | Reserve a place in ROM to be filled by some other tool, for example, a checksum calculated by <code>ielftool</code> . |
| <code>--preconfig</code>             | Reads the specified file before reading the linker configuration file                                                 |
| <code>--printf_multibytes</code>     | Makes the <code>printf</code> formatter support multibytes                                                            |
| <code>--redirect</code>              | Redirects a reference to a symbol to another symbol                                                                   |
| <code>--remarks</code>               | Enables remarks                                                                                                       |
| <code>--scanf_multibytes</code>      | Makes the <code>scanf</code> formatter support multibytes                                                             |
| <code>--search</code>                | Specifies more directories to search for object and library files                                                     |
| <code>--silent</code>                | Sets silent operation                                                                                                 |
| <code>--stack_usage_control</code>   | Specifies a stack usage control file                                                                                  |
| <code>--strip</code>                 | Removes debug information from the executable image                                                                   |
| <code>--text_out</code>              | Specifies the encoding for text output files                                                                          |
| <code>--threaded_lib</code>          | Configures the runtime library for use with threads                                                                   |
| <code>--timezone_lib</code>          | Enables the time zone and daylight savings time functionality in the library                                          |

Table 26: Linker options summary (Continued)

| Command line option                        | Description                                                                         |
|--------------------------------------------|-------------------------------------------------------------------------------------|
| <code>--use_full_std_template_names</code> | Enables full names for standard C++ templates                                       |
| <code>--utf8_text_in</code>                | Uses the UTF-8 encoding for text input files                                        |
| <code>--version</code>                     | Sends version information to the console and then exits                             |
| <code>--vfe</code>                         | Controls Virtual Function Elimination                                               |
| <code>--warnings_affect_exit_code</code>   | Warnings affects exit code                                                          |
| <code>--warnings_are_errors</code>         | Warnings are treated as errors                                                      |
| <code>--whole_archive</code>               | Treats every object file in the archive as if it was specified on the command line. |

Table 26: Linker options summary (Continued)

## Descriptions of linker options

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

To comply with the RL78 ABI, the compiler generates assembler labels for symbol and function names by prefixing an underscore. You must remember to add this extra underscore when you refer to C symbols in any of the linker options, such as `--define_symbol` and `--redirect`, or in directives in the linker configuration file, such as `define symbol`. For example, `main` must be written as `_main`.



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.

### `--call_graph`

|             |                                                                                                                                                                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--call_graph {filename directory}</code>                                                                                                                                                                                                           |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 251.                                                                                                                                                                        |
| Description | Use this option to produce a call graph file. If no filename extension is specified, the extension <code>cgx</code> is used. This option can only be used once on the command line.<br><br>Using this option enables stack usage analysis in the linker. |
| See also    | <i>Stack usage analysis</i> , page 97                                                                                                                                                                                                                    |



**Project>Options>Linker>Advanced>Call graph output (XML)**

## --config

|             |                                                                                                                                                                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--config filename</code>                                                                                                                                                                                                                                   |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 251.                                                                                                                                                                                |
| Description | Use this option to specify the configuration file to be used by the linker (the default filename extension is <code>icf</code> ). If no configuration file is specified, a default configuration is used. This option can only be used once on the command line. |
| See also    | The chapter <i>The linker configuration file</i> .                                                                                                                                                                                                               |



**Project>Options>Linker>Config>Linker configuration file**

## --config\_def

|                       |                                                                                                                                                                                                                                                                           |               |                                                              |                       |                                                 |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|--------------------------------------------------------------|-----------------------|-------------------------------------------------|
| Syntax                | <code>--config_def symbol=constant_value</code>                                                                                                                                                                                                                           |               |                                                              |                       |                                                 |
| Parameters            | <table> <tr> <td><i>symbol</i></td> <td>The name of the symbol to be used in the configuration file.</td> </tr> <tr> <td><i>constant_value</i></td> <td>The constant value of the configuration symbol.</td> </tr> </table>                                               | <i>symbol</i> | The name of the symbol to be used in the configuration file. | <i>constant_value</i> | The constant value of the configuration symbol. |
| <i>symbol</i>         | The name of the symbol to be used in the configuration file.                                                                                                                                                                                                              |               |                                                              |                       |                                                 |
| <i>constant_value</i> | The constant value of the configuration symbol.                                                                                                                                                                                                                           |               |                                                              |                       |                                                 |
| Description           | Use this option to define a constant configuration symbol to be used in the configuration file. This option has the same effect as the <code>define symbol</code> directive in the linker configuration file. This option can be used more than once on the command line. |               |                                                              |                       |                                                 |
| See also              | <code>--define_symbol</code> , page 297 and <i>Interaction between ILINK and the application</i> , page 114.                                                                                                                                                              |               |                                                              |                       |                                                 |



**Project>Options>Linker>Config>Defined symbols for configuration file**

## --config\_search

|             |                                                                                                                                                        |             |                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|----------------------------------------------------------------------------------------------|
| Syntax      | <code>--config_search path</code>                                                                                                                      |             |                                                                                              |
| Parameters  | <table> <tr> <td><i>path</i></td> <td>A path to a directory where the linker should search for linker configuration include files.</td> </tr> </table> | <i>path</i> | A path to a directory where the linker should search for linker configuration include files. |
| <i>path</i> | A path to a directory where the linker should search for linker configuration include files.                                                           |             |                                                                                              |

**Description** Use this option to specify more directories to search for files when processing an `include` directive in a linker configuration file.

By default, the linker searches for configuration include files only in the system configuration directory. To specify more than one search directory, use this option for each path.

**See also** *include directive*, page 436.



To set this option, use **Project>Options>Linker>Extra Options**.

## **--cpp\_init\_routine**

**Syntax** `--cpp_init_routine routine`

**Parameters** *routine* A user-defined C++ dynamic initialization routine.

**Description** When using the IAR C/C++ compiler and the standard library, C++ dynamic initialization is handled automatically. In other cases you might need to use this option.

If any sections with the section type `INIT_ARRAY` or `PREINIT_ARRAY` are included in your application, the C++ dynamic initialization routine is considered to be needed. By default, this routine is named `__iar_cstart_call_ctors` and is called by the startup code in the standard library. Use this option if you require another routine to handle the initialization, for instance if you are not using the standard library.



To set this option, use **Project>Options>Linker>Extra Options**.

## **--debug\_lib**

**Syntax** `--debug_lib`

**Description** Use this option to enable C-SPY emulated I/O.

**See also** *Briefly about C-SPY emulated I/O*, page 123.



**Project>Options>Linker>Library>Include C-SPY debugging support**



## --default\_to\_complex\_ranges

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--default_to_complex_ranges</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description | <p>Normally, if <code>initialize</code> directives in a linker configuration file do not specify <code>simple_ranges</code> or <code>complex_ranges</code>, the linker uses <code>simple_ranges</code> if the associated section placement directives use single range regions.</p> <p>Use this option to make the linker always use <code>complex_ranges</code> by default. This was the behavior of the linker before the introduction of <code>simple_ranges</code> and <code>complex_ranges</code>.</p> |
| See also    | <i>initialize directive</i> , page 418.                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |



**Project>Options>Linker>Extra Options**

## --define\_symbol

|                       |                                                                                                                                                                                                                                                                    |               |                                                                      |                       |                                   |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|----------------------------------------------------------------------|-----------------------|-----------------------------------|
| Syntax                | <code>--define_symbol symbol=constant_value</code>                                                                                                                                                                                                                 |               |                                                                      |                       |                                   |
| Parameters            | <table> <tr> <td><i>symbol</i></td> <td>The name of the constant symbol that can be used by the application.</td> </tr> <tr> <td><i>constant_value</i></td> <td>The constant value of the symbol.</td> </tr> </table>                                              | <i>symbol</i> | The name of the constant symbol that can be used by the application. | <i>constant_value</i> | The constant value of the symbol. |
| <i>symbol</i>         | The name of the constant symbol that can be used by the application.                                                                                                                                                                                               |               |                                                                      |                       |                                   |
| <i>constant_value</i> | The constant value of the symbol.                                                                                                                                                                                                                                  |               |                                                                      |                       |                                   |
| Description           | <p>Use this option to define a constant symbol, that is a label, that can be used by your application. This option can be used more than once on the command line.</p> <p><b>Note:</b> This option is different from the <code>define symbol</code> directive.</p> |               |                                                                      |                       |                                   |
| See also              | <i>--config_def</i> , page 295 and <i>Interaction between ILINK and the application</i> , page 114.                                                                                                                                                                |               |                                                                      |                       |                                   |



**Project>Options>Linker>#define>Defined symbols**

## --dependencies

|                          |                                                                                                      |                          |                               |
|--------------------------|------------------------------------------------------------------------------------------------------|--------------------------|-------------------------------|
| Syntax                   | <code>--dependencies [= [i m]] {filename directory}</code>                                           |                          |                               |
| Parameters               | <table> <tr> <td><code>i</code> (default)</td> <td>Lists only the names of files</td> </tr> </table> | <code>i</code> (default) | Lists only the names of files |
| <code>i</code> (default) | Lists only the names of files                                                                        |                          |                               |

`m` Lists in makefile style

See also *Rules for specifying a filename or directory as parameters*, page 251.

**Description** Use this option to make the linker list the names of the linker configuration, object, and library files opened for input into a file with the default filename extension `i`.

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

```
c:\myproject\foo.o
d:\myproject\bar.o
```

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 output file, a colon, a space, and the name of an input file. For example:

```
a.out: c:\myproject\foo.o
a.out: d:\myproject\bar.o
```



This option is not available in the IDE.

## **--diag\_error**

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

**Parameters**

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

**Description** Use this option to reclassify certain diagnostic messages as errors. An error indicates a problem of such severity that an executable image will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.



**Project>Options>Linker>Diagnostics>Treat these as errors**

## --diag\_remark

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_remark=tag[, tag, ...]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                        |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | The number of a diagnostic message, for example, the message number <code>Go109</code> |
| Description | <p>Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a construction that may cause strange behavior in the executable image.</p> <p><b>Note:</b> Not all diagnostic messages can be reclassified. This option may be used more than once on the command line.</p> <p><b>Note:</b> By default, remarks are not displayed—use the <code>--remarks</code> option to display them.</p> |                                                                                        |



**Project>Options>Linker>Diagnostics>Treat these as remarks**

## --diag\_suppress

|             |                                                                                                                                                                                                                                        |                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_suppress=tag[, tag, ...]</code>                                                                                                                                                                                           |                                                                                        |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                             | The number of a diagnostic message, for example, the message number <code>Pa180</code> |
| Description | <p>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.</p> <p><b>Note:</b> Not all diagnostic messages can be reclassified.</p> |                                                                                        |



**Project>Options>Linker>Diagnostics>Suppress these diagnostics**

## --diag\_warning

|             |                                                                                                                                                                                                                                                                          |                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--diag_warning=tag[, tag, ...]</code>                                                                                                                                                                                                                              |                                                                                        |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                               | The number of a diagnostic message, for example, the message number <code>Li004</code> |
| Description | Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the linker to stop before linking is completed. This option may be used more than once on the command line. |                                                                                        |

**Note:** Not all diagnostic messages can be reclassified.



**Project>Options>Linker>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 251.                                                                                                                                                                   |  |
| Description | Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why. |  |

This option cannot be given together with other options.



This option is not available in the IDE.

## --enable\_stack\_usage

|             |                                                                                                                                      |  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--enable_stack_usage</code>                                                                                                    |  |
| Description | Use this option to enable stack usage analysis. If a linker map file is produced, a stack usage chapter is included in the map file. |  |

**Note:** If you use at least one of the `--stack_usage_control` or `--call_graph` options, stack usage analysis is automatically enabled.

See also *Stack usage analysis*, page 97.



### Project>Options>Linker>Advanced>Enable stack usage analysis

## --entry

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                       |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| Syntax      | <code>--entry <i>symbol</i></code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                       |
| Parameters  | <i>symbol</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | The name of the symbol to be treated as a root symbol and start label |
| Description | <p>Use this option to make a symbol be treated as a root symbol and the start label of the application. This is useful for loaders. If this option is not used, the default start symbol is <code>__iar_program_start</code>. A root symbol is kept whether or not it is referenced from the rest of the application, provided its module is included. A module in an object file is always included but a module part of a library is only included if needed.</p> <p><b>Note:</b> The label referred to must be available in your application. You must also make sure that the reset vector refers to the new start label, for example <code>--redirect __iar_program_start=_myStartLabel</code>.</p> |                                                                       |



### Project>Options>Linker>Library>Override default program entry

## --entry\_list\_in\_address\_order

|             |                                                                                                                        |  |
|-------------|------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--entry_list_in_address_order</code>                                                                             |  |
| Description | Use this option to generate an additional entry list in the map file. This entry list will be sorted in address order. |  |



To set this option use **Project>Options>Linker>Extra Options**.

## --error\_limit

|            |                                     |                                                                                                                  |
|------------|-------------------------------------|------------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>--error_limit=<i>n</i></code> |                                                                                                                  |
| Parameters | <i>n</i>                            | The number of errors before the linker stops linking. <i>n</i> must be a positive integer. 0 indicates no limit. |

**Description** Use the `--error_limit` option to specify the number of errors allowed before the linker stops the linking. By default, 100 errors are allowed.



This option is not available in the IDE.

## **--export\_builtin\_config**

**Syntax** `--export_builtin_config filename`

**Parameters** See *Rules for specifying a filename or directory as parameters*, page 251.

**Description** Exports the configuration used by default to a file.



This option is not available in the IDE.

## **-f**

**Syntax** `-f filename`

**Parameters** See *Rules for specifying a filename or directory as parameters*, page 251.

**Description** Use this option to make the linker read command line options from the named file, with the default filename extension `.xcl`.

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

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

**See also** `--f`, page 302.



To set this option, use **Project>Options>Linker>Extra Options**.

## **--f**

**Syntax** `--f filename`

**Parameters** See *Rules for specifying a filename or directory as parameters*, page 251.

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Use this option to make the linker read command line options from the named file, with the default filename extension <code>.xcl</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.</p> <p>Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.</p> <p>If you use the linker option <code>--dependencies</code>, extended command line files specified using <code>--f</code> will generate a dependency, but those specified using <code>-f</code> will not generate a dependency.</p> |
| See also    | <p><code>--dependencies</code>, page 265 and <code>-f</code>, page 276.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |



To set this option, use **Project>Options>Linker>Extra Options**.

## --force\_output

|             |                                                                                         |
|-------------|-----------------------------------------------------------------------------------------|
| Syntax      | <code>--force_output</code>                                                             |
| Description | Use this option to produce an output executable image regardless of any linking errors. |



To set this option, use **Project>Options>Linker>Extra Options**

## --image\_input

|            |                                                                     |                                                                                   |
|------------|---------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| Syntax     | <code>--image_input filename [,symbol,[section[,alignment]]]</code> |                                                                                   |
| Parameters | <i>filename</i>                                                     | The pure binary file containing the raw image you want to link                    |
|            | <i>symbol</i>                                                       | The symbol which the binary data can be referenced with.                          |
|            | <i>section</i>                                                      | The section where the binary data will be placed. Default is <code>.text</code> . |
|            | <i>alignment</i>                                                    | The alignment of the section. Default is 1.                                       |

**Description** Use this option to link pure binary files in addition to the ordinary input files. The file's entire contents are placed in the section, which means it can only contain pure binary data.

**Note:** Just as for sections from object files, sections created by using the `--image_input` option are not included unless actually needed. You can either specify a symbol in the option and reference this symbol in your application (or use a `--keep` option), or you can specify a section name and use the `keep` directive in a linker configuration file to ensure that the section is included.

**Example** `--image_input bootstrap.abs,Bootstrap,CSTARTUPCODE,4`

The contents of the pure binary file `bootstrap.abs` are placed in the section `CSTARTUPCODE`. The section where the contents are placed is 4-byte aligned and will only be included if your application (or the command line option `--keep`) includes a reference to the symbol `Bootstrap`.

**See also** `--keep`, page 304.



**Project>Options>Linker>Input>Raw binary image**

## --inline

**Syntax** `--inline`

**Description** Some routines are so small that they can fit in the space of the instruction that calls the routine. Use this option to make the linker replace the call of a routine with the body of the routine, where applicable.

**See also** *Small function inlining*, page 120.



**Project>Options>Linker>Optimizations>Inline small routines**

## --keep

**Syntax** `--keep symbol`

**Parameters** `symbol` The name of the symbol to be treated as a root symbol



**Description** Normally, the linker keeps a symbol only if it is needed by your application. Use this option to make a symbol always be included in the final application.



**Project>Options>Linker>Input>Keep symbols**

## --log

**Syntax** `--log topic[,topic,...]`

**Parameters** `topic` can be one of:

|                               |                                                                                                                                                                                                                               |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>call_graph</code>       | Lists the call graph as seen by stack usage analysis.                                                                                                                                                                         |
| <code>initialization</code>   | Lists copy batches and the compression selected for each batch.                                                                                                                                                               |
| <code>libraries</code>        | Lists all decisions made by the automatic library selector. This might include extra symbols needed ( <code>--keep</code> ), redirections ( <code>--redirect</code> ), as well as which runtime libraries that were selected. |
| <code>modules</code>          | Lists each module that is selected for inclusion in the application, and which symbol that caused it to be included.                                                                                                          |
| <code>redirects</code>        | Lists redirected symbols.                                                                                                                                                                                                     |
| <code>sections</code>         | Lists each symbol and section fragment that is selected for inclusion in the application, and the dependence that caused it to be included.                                                                                   |
| <code>unused_fragments</code> | Lists those section fragments that were not included in the application.                                                                                                                                                      |

**Description** Use this option to make the linker log information to `stdout`. The log information can be useful for understanding why an executable image became the way it is.

**See also** `--log_file`, page 306.



**Project>Options>Linker>List>Generate log**

## --log\_file

|             |                                                                                   |
|-------------|-----------------------------------------------------------------------------------|
| Syntax      | <code>--log_file filename</code>                                                  |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 251. |
| Description | Use this option to direct the log output to the specified file.                   |
| See also    | <code>--log</code> , page 305.                                                    |



**Project>Options>Linker>List>Generate log**

## --mangled\_names\_in\_messages

|             |                                                                                                                                                                                                                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--mangled_names_in_messages</code>                                                                                                                                                                                                                                                      |
| Description | Use this option to produce both mangled and unmangled names for C/C++ symbols in messages. Mangling is a technique used for mapping a complex C name or a C++ name—for example, for overloading—into a simple name. For example, <code>void h(int, char)</code> becomes <code>_Z1hic</code> . |



This option is not available in the IDE.

## --manual\_dynamic\_initialization

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--manual_dynamic_initialization</code>                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description | Normally, dynamic initialization (typically initialization of C++ objects with static storage duration) is performed automatically during application startup. If you use <code>--manual_dynamic_initialization</code> , you must call <code>__iar_dynamic_initialization</code> at some later point for this initialization to be done.<br><br>The function <code>__iar_dynamic_initialization</code> is declared in the header file <code>iar_dynamic_init.h</code> . |



To set this option use **Project>Options>Linker>Extra Options**.

## --map

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--map {filename directory}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Description | <p>Use this option to produce a linker memory map file. The map file has the default filename extension <code>map</code>. The map file contains:</p> <ul style="list-style-type: none"> <li>● Linking summary in the map file header which lists the version of the linker, the current date and time, and the command line that was used.</li> <li>● Runtime attribute summary which lists runtime attributes.</li> <li>● Placement summary which lists each section/block in address order, sorted by placement directives.</li> <li>● Initialization table layout which lists the data ranges, packing methods, and compression ratios.</li> <li>● Module summary which lists contributions from each module to the image, sorted by directory and library.</li> <li>● Entry list which lists all public and some local symbols in alphabetical order, indicating which module they came from.</li> <li>● Some of the bytes might be reported as <i>shared</i>.</li> </ul> <p>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 also sometimes 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.</p> <p>This option can only be used once on the command line.</p> |



**Project>Options>Linker>List>Generate linker map file**

## --merge\_duplicate\_sections

|             |                                                                                                                                                                                                                                                                                                |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--merge_duplicate_sections</code>                                                                                                                                                                                                                                                        |
| Description | <p>Use this option to keep only one copy of equivalent read-only sections.</p> <p><b>Note:</b> This can cause different functions or constants to have the same address, so an application that depends on the addresses being different will not work correctly with this option enabled.</p> |

See also

*Duplicate section merging*, page 120.



**Project>Options>Linker>Optimizations>Merge duplicate sections**

## **--no\_bom**

Syntax

--no\_bom

Description

Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

See also

*--text\_out*, page 316 and *Text encodings*, page 244.



**Project>Options>Linker>Encodings>Text output file encoding**

## **--no\_entry**

Syntax

--no\_entry

Description

Use this option to set the entry point field to zero for produced ELF files.



**Project>Options>Linker>Library>Override default program entry**

## **--no\_fragments**

Syntax

--no\_fragments

Description

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and further minimize the size of the executable image. Use this option to disable the removal of fragments of sections, instead including or not including each section in its entirety, usually resulting in a larger application.

See also

*Keeping symbols and sections*, page 109.



To set this option, use **Project>Options>Linker>Extra Options**

## --no\_free\_heap

Syntax `--no_free_heap`

Description Use this option to use the smallest possible heap implementation. Because this heap does not support `free` or `realloc`, it is only suitable for applications that in the startup phase allocate heap memory for various buffers, etc, and for applications that never deallocate memory.



**Project>Options>General Options>Library Options 2>Heap selection**

## --no\_inline

Syntax `--no_inline func[, func...]`

Parameters *func* The name of a function symbol

Description Use this option to exclude some functions from small function inlining.

See also `--inline`, page 304.



To set this option, use **Project>Options>Linker>Extra Options**.

## --no\_library\_search

Syntax `--no_library_search`

Description Use this option to disable the automatic runtime library search. This option turns off the automatic inclusion of the correct standard libraries. This is useful, for example, if the application needs a user-built standard library, etc.

**Note:** The option disables all steps of the automatic library selection, some of which might need to be reproduced if you are using the standard libraries. Use the `--log libraries` linker option together with automatic library selection enabled to determine which the steps are.



**Project>Options>Linker>Library>Automatic runtime library selection**

## **--no\_locals**

Syntax `--no_locals`

Description Use this option to remove local symbols from the ELF executable image.

**Note:** This option does not remove any local symbols from the DWARF information in the executable image.



**Project>Options>Linker>Output**

## **--no\_range\_reservations**

Syntax `--no_range_reservations`

Description Normally, the linker reserves any ranges used by absolute symbols with a non-zero size, excluding them from consideration for `place in` commands.

When this option is used, these reservations are disabled, and the linker is free to place sections in such a way as to overlap the extent of absolute symbols.



To set this option, use **Project>Options>Linker>Extra Options**.

## **--no\_remove**

Syntax `--no_remove`

Description When this option is used, unused sections are not removed. In other words, each module that is included in the executable image contains all its original sections.

See also *Keeping symbols and sections*, page 109.



To set this option, use **Project>Options>Linker>Extra Options**.

## **--no\_vfe**

Syntax `--no_vfe`

Description Use this option to disable the Virtual Function Elimination optimization. All virtual functions in all classes with at least one instance will be kept, and Runtime Type

Information data will be kept for all polymorphic classes. Also, no warning message will be issued for modules that lack VFE information.

See also

`--vfe`, page 318 and *Virtual function elimination*, page 119.



To set related options, choose:

**Project>Options>Linker>Optimizations>PerformC++ Virtual Function Elimination**

## **--no\_warnings**

Syntax

`--no_warnings`

Description

By default, the linker 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.

## **--only\_stdout**

Syntax

`--only_stdout`

Description

Use this option to make the linker use the standard output stream (`stdout`), and messages that are normally directed to the error output stream (`stderr`).



This option is not available in the IDE.

## --output, -o

|             |                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--output {filename directory}</code><br><code>-o {filename directory}</code>                                                                                                                                                                             |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 251.                                                                                                                                                                              |
| Description | By default, the object executable image produced by the linker is located in a file with the name <code>a.out</code> . Use this option to explicitly specify a different output filename, which by default will have the filename extension <code>out</code> . |



**Project>Options>Linker>Output>Output file**

## --place\_holder

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                  |             |                                 |                |                                                    |                  |                                    |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|----------------------------------|-------------|---------------------------------|----------------|----------------------------------------------------|------------------|------------------------------------|
| Syntax           | <code>--place_holder symbol[,size[,section[,alignment]]]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                  |             |                                 |                |                                                    |                  |                                    |
| Parameters       | <table> <tr> <td><i>symbol</i></td> <td>The name of the symbol to create</td> </tr> <tr> <td><i>size</i></td> <td>Size in ROM. Default is 4 bytes</td> </tr> <tr> <td><i>section</i></td> <td>Section name to use. Default is <code>.text</code></td> </tr> <tr> <td><i>alignment</i></td> <td>Alignment of section. Default is 1</td> </tr> </table>                                                                                                                                                                                                                                                                                            | <i>symbol</i> | The name of the symbol to create | <i>size</i> | Size in ROM. Default is 4 bytes | <i>section</i> | Section name to use. Default is <code>.text</code> | <i>alignment</i> | Alignment of section. Default is 1 |
| <i>symbol</i>    | The name of the symbol to create                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |               |                                  |             |                                 |                |                                                    |                  |                                    |
| <i>size</i>      | Size in ROM. Default is 4 bytes                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |               |                                  |             |                                 |                |                                                    |                  |                                    |
| <i>section</i>   | Section name to use. Default is <code>.text</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |               |                                  |             |                                 |                |                                                    |                  |                                    |
| <i>alignment</i> | Alignment of section. Default is 1                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |               |                                  |             |                                 |                |                                                    |                  |                                    |
| Description      | <p>Use this option to reserve a place in ROM to be filled by some other tool, for example, a checksum calculated by <code>ie1ftool</code>. Each use of this linker option results in a section with the specified name, size, and alignment. The symbol can be used by your application to refer to the section.</p> <p><b>Note:</b> Like any other section, sections created by the <code>--place_holder</code> option will only be included in your application if the section appears to be needed. The <code>--keep</code> linker option, or the <code>keep</code> linker directive can be used for forcing such section to be included.</p> |               |                                  |             |                                 |                |                                                    |                  |                                    |
| See also         | <i>IAR utilities</i> , page 457.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |               |                                  |             |                                 |                |                                                    |                  |                                    |



To set this option, use **Project>Options>Linker>Extra Options**



## --preconfig

|             |                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--preconfig filename</code>                                                                        |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 251.                        |
| Description | Use this option to make the linker read the specified file before reading the linker configuration file. |



To set this option, use **Project>Options>Linker>Extra Options**.

## --printf\_multibytes

|             |                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--printf_multibytes</code>                                                                                  |
| Description | Use this option to make the linker automatically select a <code>printf</code> formatter that supports multibytes. |



**Project>Options>General Options>Library options 1>Printf formatter**

## --redirect

|                          |                                                                                                                                                                                             |                          |                               |                        |                                    |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|-------------------------------|------------------------|------------------------------------|
| Syntax                   | <code>--redirect from_symbol=to_symbol</code>                                                                                                                                               |                          |                               |                        |                                    |
| Parameters               | <table> <tr> <td><code>from_symbol</code></td> <td>The name of the source symbol</td> </tr> <tr> <td><code>to_symbol</code></td> <td>The name of the destination symbol</td> </tr> </table> | <code>from_symbol</code> | The name of the source symbol | <code>to_symbol</code> | The name of the destination symbol |
| <code>from_symbol</code> | The name of the source symbol                                                                                                                                                               |                          |                               |                        |                                    |
| <code>to_symbol</code>   | The name of the destination symbol                                                                                                                                                          |                          |                               |                        |                                    |
| Description              | Use this option to change references to an external symbol so that they refer to another symbol.                                                                                            |                          |                               |                        |                                    |

**Note:** Redirection will normally not affect references within a module.



To set this option, use **Project>Options>Linker>Extra Options**

## --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 linker does not generate remarks. Use this option to make the linker generate remarks. |
| See also    | <i>Severity levels</i> , page 247.                                                                                                                                                                                                                                |



**Project>Options>Linker>Diagnostics>Enable remarks**

## --scanf\_multibytes

|             |                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------|
| Syntax      | --scanf_multibytes                                                                                               |
| Description | Use this option to make the linker automatically select a <code>scanf</code> formatter that supports multibytes. |



**Project>Options>General Options>Library options 1>Scanf formatter**

## --search, -L

|             |                                                                                                                                                                                                                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --search <i>path</i><br>-L <i>path</i>                                                                                                                                                                                                                                              |
| Parameters  | <i>path</i> A path to a directory where the linker should search for object and library files.                                                                                                                                                                                      |
| Description | Use this option to specify more directories for the linker to search for object and library files in.<br><br>By default, the linker searches for object and library files only in the working directory. Each use of this option on the command line adds another search directory. |
| See also    | <i>The linking process in detail</i> , page 89.                                                                                                                                                                                                                                     |



This option is not available in the IDE.

## --silent

Syntax `--silent`

Description By default, the linker issues introductory messages and a final statistics report. Use this option to make the linker 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.

## --stack\_usage\_control

Syntax `--stack_usage_control=filename`

Parameters See *Rules for specifying a filename or directory as parameters*, page 251.

Description Use this option to specify a stack usage control file. This file controls stack usage analysis, or provides more stack usage information for modules or functions. You can use this option multiple times to specify multiple stack usage control files. If no filename extension is specified, the extension `suc` is used.

Using this option enables stack usage analysis in the linker.

See also *Stack usage analysis*, page 97.



**Project>Options>Linker>Advanced>Control file**

## --strip

Syntax `--strip`

Description By default, the linker retains the debug information from the input object files in the output executable image. Use this option to remove that information.



To set related options, choose:

**Project>Options>Linker>Output>Include debug information in output**

## --text\_out

Syntax `--text_out {utf8|utf16le|utf16be|locale}`

### Parameters

|                      |                                        |
|----------------------|----------------------------------------|
| <code>utf8</code>    | Uses the UTF-8 encoding                |
| <code>utf16le</code> | Uses the UTF-16 little-endian encoding |
| <code>utf16be</code> | Uses the UTF-16 big-endian encoding    |
| <code>locale</code>  | Uses the system locale encoding        |

### Description

Use this option to specify the encoding to be used when generating a text output file. The default for the linker list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM). If you want text output in UTF-8 encoding without BOM, you can use the option `--no_bom` as well.

### See also

`--no_bom`, page 308 and *Text encodings*, page 244.



**Project>Options>Linker>Encodings>Text output file encoding**

## --threaded\_lib

Syntax `--threaded_lib`

### Description

Use this option to automatically configure the runtime library for use with threads.



**Project>Options>General Options>Library Configuration>Enable thread support in library**

## --timezone\_lib

Syntax `--timezone_lib`

### Description

Use this option to enable the time zone and daylight savings time functionality in the DLIB library.

**Note:** You need to implement the time zone functionality.

### See also

`__getzone`, page 148.



To set this option, use **Project>Option>Linker>Extra Options**.

## --use\_full\_std\_template\_names

Syntax

--use\_full\_std\_template\_names

Description

In the unmangled names of C++ entities, the linker by default uses shorter names for some classes. For example, "std::string" instead of "std::basic\_string<char, std::char\_traits<char>, std::allocator<char>>". Use this option to make the linker instead use the full, unabbreviated names.



This option is not available in the IDE.

## --utf8\_text\_in

Syntax

--utf8\_text\_in

Description

Use this option to specify that the linker shall use the UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).

**Note:** This option does not apply to source files.

See also

*Text encodings*, page 244.



**Project>Options>Linker>Encodings>Default input file encoding**

## --version

Syntax

--version

Description

Use this option to make the linker send version information to the console and then exit.



This option is not available in th IDE.

## --vfe


|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                             |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--vfe=[forced]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                             |
| Parameters  | <code>forced</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | Performs Virtual Function Elimination even if one or more modules lack the needed virtual function elimination information. |
| Description | <p>By default, Virtual Function Elimination is always performed but requires that all object files contain the necessary virtual function elimination information. Use <code>--vfe=forced</code> to perform Virtual Function Elimination even if one or more modules do not have the necessary information.</p> <p>Forcing the use of Virtual Function Elimination can be unsafe if some of the modules that lack the needed information perform virtual function calls or use dynamic Runtime Type Information.</p> |                                                                                                                             |
| See also    | <code>--no_vfe</code> , page 310 and <i>Virtual function elimination</i> , page 119.                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                             |



To set related options, choose:

**Project>Options>Linker>Optimizations>Perform C++ Virtual Function Elimination**

## --warnings\_affect\_exit\_code

|             |                                                                                                                                                                                     |                                          |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------|
| Syntax      | <code>--warnings_affect_exit_code</code>                                                                                                                                            |                                          |
| Description | <p>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.</p> |                                          |
|             |                                                                                                  | This option is not available in the IDE. |

## --warnings\_are\_errors

|             |                                                                                                                                                                                                                      |  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--warnings_are_errors</code>                                                                                                                                                                                   |  |
| Description | <p>Use this option to make the linker treat all warnings as errors. If the linker encounters an error, no executable image is generated. Warnings that have been changed into remarks are not treated as errors.</p> |  |

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

See also

`--diag_warning`, page 269 and `--diag_warning`, page 300.



**Project>Options>Linker>Diagnostics>Treat all warnings as errors**

## **--whole\_archive**

Syntax

`--whole_archive filename`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 251.

Description

Use this option to make the linker treat every object file in the archive as if it was specified on the command line. This is useful when an archive contains root content that is always included from an object file (filename extension `o`), but only included from an archive if some entry from the module is referred to.

Example

If `archive.a` contains the object files `file1.o`, `file2.o`, and `file3.o`, using `--whole_archive archive.a` is equivalent to specifying `file1.o file2.o file3.o`.

See also

*Keeping modules*, page 109.



To set this option, use **Project>Options>Linker>Extra Options**





# 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` or the `__packed` data type attribute.

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

alignment requirements. 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 331.

**Note:** With the `#pragma data_alignment` directive, you can increase the alignment demands on specific variables.

See also the Standard C file `stdalign.h`.

## ALIGNMENT ON THE RL78 MICROCONTROLLER

The RL78 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 RL78 microcontroller can read the data. In this case, the word assembler instruction `MOVW` will be used.

**Note:** Because all arrays and string literals receive alignment 2, the compiler might add pad bytes at the end if their size in bytes is odd.

---

## Basic data types—integer types

The compiler supports both all Standard C basic data types and some additional types.

These topics are covered:

- *Integer types—an overview*, page 322
- *Bool*, page 323
- *The enum type*, page 323
- *The char type*, page 323
- *The wchar\_t type*, page 324
- *The char16\_t type*, page 324
- *The char32\_t type*, page 324
- *Bitfields*, page 324

## 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         |

*Table 27: Integer types*

| Data type          | Size    | Range                   | Alignment |
|--------------------|---------|-------------------------|-----------|
| signed char        | 8 bits  | -128 to 127             | 1         |
| unsigned char      | 8 bits  | 0 to 255                | 1         |
| signed short       | 16 bits | -32768 to 32767         | 2         |
| unsigned short     | 16 bits | 0 to 65535              | 2         |
| signed int         | 16 bits | -32768 to 32767         | 2         |
| unsigned int       | 16 bits | 0 to 65535              | 2         |
| signed long        | 32 bits | $-2^{31}$ to $2^{31}-1$ | 2         |
| unsigned long      | 32 bits | 0 to $2^{32}-1$         | 2         |
| signed long long   | 64 bits | $-2^{63}$ to $2^{63}-1$ | 2         |
| unsigned long long | 64 bits | 0 to $2^{64}-1$         | 2         |

Table 27: Integer types (Continued)

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

## BOOL

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

## THE ENUM TYPE

The compiler will use the smallest type required to hold `enum` constants, preferring signed rather than unsigned.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

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

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

See also the C++ `enum struct` syntax.

## 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:** The library is compiled with the `char` type as unsigned.

## THE WCHAR\_T TYPE

The `wchar_t` data type is 4 bytes and the encoding used for it is UTF-32.

## THE CHAR16\_T TYPE

The `char16_t` data type is 2 bytes and the encoding used for it is UTF-16.

## THE CHAR32\_T TYPE

The `char32_t` data type is 4 bytes and the encoding used for it is UTF-32.

## BITFIELDS

In Standard C, `int`, `signed int`, and `unsigned int` can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (`char`, `short`, `int`, etc) results in a signed or unsigned bitfield.

In the IAR C/C++ Compiler for RL78, 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.

If you are using the V2 calling convention as the default calling convention for the project, each bitfield is placed in the next container of its base type that has enough available bits to accommodate the bitfield. Within each container, the bitfield is placed in the first available byte or bytes, taking the byte order into account. Note that containers can overlap if needed, as long as they are suitably aligned for their type.

In addition, the compiler supports an alternative bitfield allocation strategy (disjoint types), where bitfield containers of different types are not allowed to overlap. Using this allocation strategy, each bitfield is placed in a new container if its type is different from that of the previous bitfield, or if the bitfield does not fit in the same container as the previous bitfield. Within each container, the bitfield is placed from the least significant bit to the most significant bit (disjoint types) or from the most significant bit to the least significant bit (reverse disjoint types). This allocation strategy will never use less space than the default allocation strategy (joined types), and can use significantly more space when mixing bitfield types.

**Note:** If you are using the V1 calling convention as the default calling convention for the project, disjoint types is the only available allocation strategy.

If the V2 calling convention is the default calling convention, you can use the `#pragma bitfields` directive to choose which bitfield allocation strategy to use, see *Bitfields*, page 324. If you use the disjoint types allocation strategy, you can also use the

directive `#pragma bitfields` to place bitfields from the most significant bit to the least significant bit in each container.

Assume this example:

```
struct BitfieldExample
{
 uint32_t a : 12;
 uint16_t b : 3;
 uint16_t c : 7;
 uint8_t d;
};
```

### The example in the joined types bitfield allocation strategy

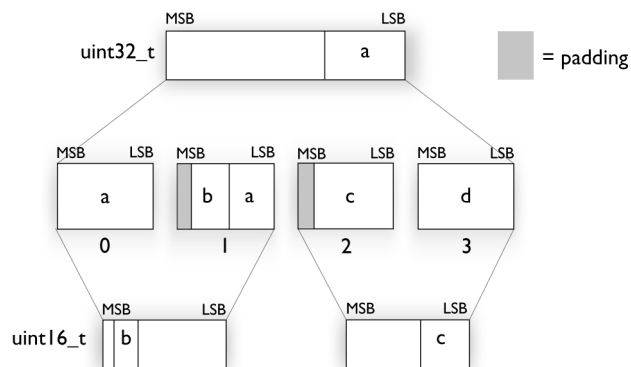
To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset 0 and puts `a` into the first and second bytes of the container.

For the second bitfield, `b`, a 16-bit container is needed and because there are still four bits free at offset 0, the bitfield is placed there.

For the third bitfield, `c`, as there is now only one bit left in the first 16-bit container, a new container is allocated at offset 2, and `c` is placed in the first byte of this container.

The fourth member, `d`, can be placed in the next available full byte, which is the byte at offset 3.

, each bitfield is allocated starting from the least significant free bit of its container to ensure that it is placed into bytes from left to right.



### 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.

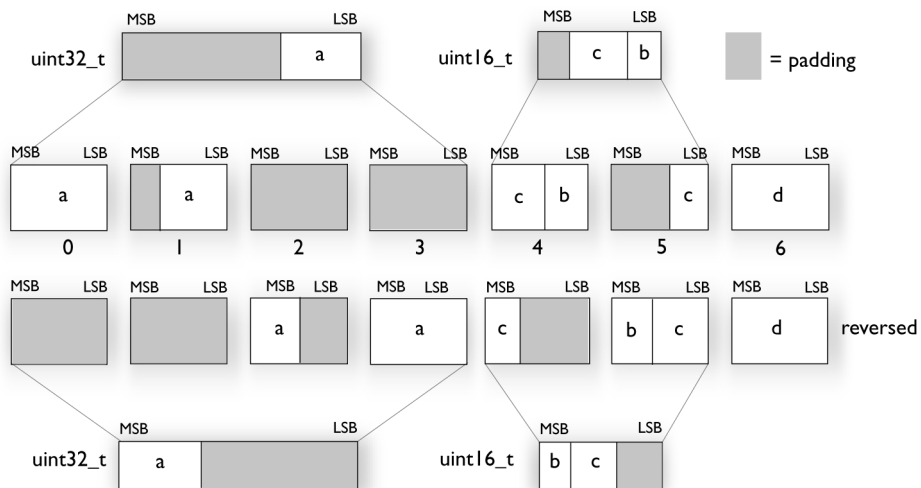
To place the second bitfield, *b*, a new container is allocated at offset 4, because the type of the bitfield is not the same as that of the previous one. *b* is placed into the least significant three bits of this container.

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 (reverse disjoint types), 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 RL78, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

| Type        | Size if double=32 | Size if double=64 |
|-------------|-------------------|-------------------|
| float       | 32 bits           | 32 bits           |
| double      | 32 bits (default) | 64 bits           |
| long double | 32 bits           | 64 bits           |

Table 28: Floating-point types

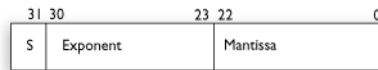
**Note:** The size of `double` and `long double` depends on the `--double={32|64}` option, see `--double`, page 266. The type `long double` uses the same precision as `double`.

## FLOATING-POINT ENVIRONMENT

Exception flags are not supported. The `feraiseexcept` function does not raise any exceptions.

### 32-BIT FLOATING-POINT FORMAT

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



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

The value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-127)} * 1.\text{Mantissa}$$

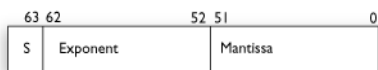
The range of the number is at least:

$$\pm 1.18\text{E}-38 \text{ to } \pm 3.39\text{E}+38$$

The precision of the float operators (+, -, \*, and /) is approximately 7 decimal digits.

### 64-BIT FLOATING-POINT FORMAT

The representation of a 64-bit floating-point number as an integer is:



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-1023)} * 1.\text{Mantissa}$$

The range of the number is at least:

$$\pm 2.23\text{E}-308 \text{ to } \pm 1.79\text{E}+308$$

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

## REPRESENTATION OF SPECIAL FLOATING-POINT NUMBERS

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

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored. The uppermost bits must be set.
- Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is subnormal, even though the number is treated as if the exponent was 1. Unlike normal numbers, subnormal numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a subnormal number is:

$$(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$$

where BIAS is 127.

---

## Pointer types

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

### FUNCTION POINTERS

The size of function pointers is always 16 or 24 bits. These function pointers are available:

| Keyword                  | Pointer size | Default in code model | Description                                  |
|--------------------------|--------------|-----------------------|----------------------------------------------|
| <code>__near_func</code> | 2 bytes      | Near                  | Can address the first 64 Kbytes of memory.   |
| <code>__far_func</code>  | 3 bytes*     | Far                   | Can address the entire 1 Mbyte memory space. |

*Table 29: Function pointers*

\* Because of alignment restrictions, 3-byte pointers take up 4 bytes in memory.



## DATA POINTERS

The size of data pointers is always 16 or 24 bits. These data pointers are available:

| Keyword             | Pointer size | Address range   | Description                                                          |
|---------------------|--------------|-----------------|----------------------------------------------------------------------|
| <code>__near</code> | 2 bytes      | 0xF0000–0xFFFFF | The highest 64 Kbytes.                                               |
| <code>__far</code>  | 3 bytes*     | 0x00000–0xFFFFF | Entire 1 Mbyte address area.                                         |
| <code>__huge</code> | 3 bytes*     | 0x00000–0xFFFFF | Entire 1 Mbyte address area. Used for objects larger than 64 Kbytes. |

Table 30: Data pointers

\* Because of alignment restrictions, 3-byte pointers take up 4 bytes in memory.

## CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a value of an integer type to a pointer of a larger type is performed by zero extension
- Casting a 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 a `__near` pointer to a `__far` or a `__huge` pointer is performed by extension with 0xF0000
- Casting a `__far` or a `__huge` pointer to a `__near` pointer is an illegal operation
- Casting a `__near_func` pointer to a `__far_func` pointer is performed by zero extension
- Casting a `__far_func` pointer to a `__near_func` pointer is an illegal operation
- Casting a `__far` pointer to a `__huge` pointer results in the same bit pattern
- Casting a `__huge` pointer to a `__far` pointer is an illegal operation.

### `size_t`

`size_t` is the unsigned integer type of the result of the `sizeof` operator. In the IAR C/C++ Compiler for RL78, the type used for `size_t` is unsigned `int`.

**ptrdiff\_t**

`ptrdiff_t` is the signed integer type of the result of subtracting two pointers. In the IAR C/C++ Compiler for RL78, the type used for `ptrdiff_t` is the signed integer variant of the `size_t` type.

Note that subtracting pointers other than default pointers could result in a smaller or larger integer type. In each case, this integer type is the signed integer variant of the corresponding `size_t` type.

**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; /* Result: -5536 */
```

**intptr\_t**

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for RL78, the type used for `intptr_t` is signed `long`.

**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**

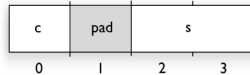
The `struct` and `union` types have the same alignment as the member with the highest alignment requirement—this alignment requirement also applies to a member that is a structure. To allow arrays of aligned structure objects, the size of a `struct` is adjusted to an even multiple of the alignment.

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

**Note:** 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.

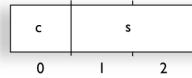
Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, 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()
```

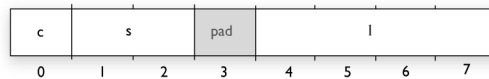
The structure `s` has this memory layout:



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 220.

---

## 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—therefore 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 RL78 are described below.

### Rules for accesses

In the IAR C/C++ Compiler for RL78, accesses to `volatile` declared objects are subject to these rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The compiler adheres to these rules for all 8-bit data types. Variables declared `volatile` and `__sfr` can be accessed using bit access unless they are also declared `__nokitaccess`.

For all other object types, only the rule that states that all accesses are preserved applies.

### DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, declare it with the `__ro_placement` attribute. See `__ro_placement`, page 348.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, define the variable like this:

```
const volatile int x @ "FLASH";
```

The compiler will generate the read/write section `FLASH`. That section should be placed in ROM and is used for manually initializing the variables 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` and located in far memory are allocated in ROM.

`const` declared `saddr` objects are allocated in RAM and initialized by the runtime system at startup. `const` declared near objects are allocated according to the option `--near_const_location`.

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 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
- Supported GCC attributes

For information about the address ranges of the different memory areas, see the chapter *Section reference*.

---

## 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 RL78 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 340. For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*.

**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 273.

### 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 *function memory attributes*:

```
__near_func, __far_func, __callt
```

Available *data memory attributes*:

```
__saddr, __near, __far, __huge, __sfr.
```

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, __no_save, __v1_call, __v2_call
```

Available *data type attributes*:

```
__no_bit_access, __packed
```

You can specify as many type attributes as required for each level of pointer indirection.

### Syntax for type attributes used on data objects

If you select the *uniform attribute syntax*, data type attributes use the same syntax rules as the type qualifiers `const` and `volatile`.

If not, data type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__near int i;
int __near j;
```



Both `i` and `j` are placed in near 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 __near * p; /* pointer to integer in near memory */
int * __near p; /* pointer in near memory */
__near int * p; /* pointer in near memory */
```

The third case is interpreted differently when uniform attribute syntax is selected. If so, it is equivalent to the first case, just as would be the case if `const` or `volatile` were used correspondingly.

In all cases, if a memory attribute is not specified, an appropriate default memory type is used, which depends on the data model in use.

Using a type definition can sometimes make the code clearer:

```
typedef __near int d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in near 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=__near
int * q2;
```

The variable `q2` is placed in near memory.

For more examples of using memory attributes, see *More examples*, page 70.

For more information about the uniform attribute syntax, see *--uniform\_attribute\_syntax*, page 305 and *--no\_uniform\_attribute\_syntax*, page 292.

### 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 inside the parentheses for function pointers, for example:

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

or

```
void (__interrupt * my_fp)(void);
```

You can also use `#pragma type_attribute` to specify the function type attributes:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

```
#pragma type attribute=__interrupt
typedef void my_fun_t(void);
my_fun_t * my_fp;
```

To declare a function pointer, use this syntax:

```
int (__near_func * fp)(double);
```

After this declaration, the function pointer `fp` points to near memory.

An easier way of specifying storage is to use type definitions:

```
typedef __near_func void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

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

## 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, __ro_placement
```

- Object attributes that can be used for functions and variables:

```
location, @, __root, __weak
```

- Object attributes that can be used for functions:

```
__intrinsic, __noreturn, 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 222. For more information about `vector`, see *vector*, page 378.

### 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>__callt</code>                                          | Controls the storage of functions                                                 |
| <code>__far</code>                                            | Controls the storage of data objects                                              |
| <code>__far_func</code>                                       | Controls the storage of functions                                                 |
| <code>__huge</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>__near</code>                                           | Controls the storage of data objects                                              |
| <code>__near_func</code>                                      | Controls the storage of functions                                                 |
| <code>__no_alloc,</code><br><code>__no_alloc16</code>         | Makes a constant available in the execution file                                  |
| <code>__no_alloc_str,</code><br><code>__no_alloc_str16</code> | Makes a string literal available in the execution file                            |
| <code>__no_bit_access</code>                                  | Inhibits the use of bit instructions when accessing individual bits of the object |
| <code>__no_init</code>                                        | Places a data object in non-volatile memory                                       |

Table 31: Extended keywords summary

| Extended keyword            | Description                                                                       |
|-----------------------------|-----------------------------------------------------------------------------------|
| <code>__noreturn</code>     | Informs the compiler that the function will not return                            |
| <code>__no_save</code>      | Specifies a function to not save and restore the workseg area that it has used    |
| <code>__packed</code>       | Decreases data type alignment to 1                                                |
| <code>__root</code>         | Ensures that a function or variable is included in the object code even if unused |
| <code>__ro_placement</code> | Places <code>const volatile</code> data in read-only memory.                      |
| <code>__saddr</code>        | Controls the storage of data objects                                              |
| <code>__sfr</code>          | Controls the storage of data objects                                              |
| <code>__v1_call</code>      | Specifies the V1 calling convention                                               |
| <code>__v2_call</code>      | Specifies the V2 calling convention                                               |
| <code>__weak</code>         | Declares a symbol to be externally weakly linked                                  |

Table 31: Extended keywords summary (Continued)

## Descriptions of extended keywords

This section gives detailed information about each extended keyword.

### `__callt`

|                     |                                                                                                                                                                                                                                                                                        |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on functions</i> , page 337.                                                                                                                                                                                                                    |
| Description         | The <code>__callt</code> memory attribute overrides the default storage of functions given by the selected code model and specifies that individual functions are called with a vectored <code>CALLT</code> instruction. The function can be placed in any <code>XLINK</code> segment. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0-0xFFFF (64 Kbytes)</li> <li>● Maximum size: 65535 bytes.</li> <li>● Pointer size: 2 bytes</li> </ul>                                                                                                                         |
| Example             | <code>__callt void myfunction(void);</code>                                                                                                                                                                                                                                            |
| See also            | <code>--generate_callt_runtime_library_calls</code> , page 277.                                                                                                                                                                                                                        |

**\_\_far**

|                     |                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on data objects</i> , page 336.                                                                                                                                                                                                                                                                                     |
| Description         | <p>The <code>__far</code> memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in far memory.</p> <p>You can also use the <code>__far</code> attribute to create a pointer explicitly pointing to an object located in the far memory.</p>                           |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0–0xFFFFF (1 Mbyte)</li> <li>● Maximum object size: 65535 bytes. An object cannot cross a 64-Kbyte boundary.</li> <li>● Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except equality comparison which is always performed on the entire 24-bit address.</li> </ul> |
| Example             | <code>__far int x;</code>                                                                                                                                                                                                                                                                                                                                  |
| See also            | <i>Memory types</i> , page 66.                                                                                                                                                                                                                                                                                                                             |


**\_\_far\_func**

|                     |                                                                                                                                                                                                                                                                                                                                                           |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on functions</i> , page 337.                                                                                                                                                                                                                                                                                       |
| Description         | <p>The <code>__far_func</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in <code>far_func</code> memory. You can also use the <code>__far_func</code> attribute to create a pointer explicitly pointing to an object located in the <code>far_func</code> memory.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0–0xFFFFF (1 Mbyte)</li> <li>● Maximum size: 65535 bytes. A function cannot cross a 64-Kbyte boundary.</li> <li>● Pointer size: 3 bytes</li> </ul>                                                                                                                                                |
| Example             | <code>__far_func void myfunction(void);</code>                                                                                                                                                                                                                                                                                                            |
| See also            | <i>Code models and memory attributes for function storage</i> , page 77.                                                                                                                                                                                                                                                                                  |

## \_\_huge

|                     |                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on data objects</i> , page 336.                                                                                                                                                                                                                                                                                                                     |
| Description         | <p>The <code>__huge</code> memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants anywhere in memory, regardless of their size.</p> <p>You can also use the <code>__huge</code> attribute to create a pointer explicitly pointing to an object located anywhere in memory, regardless of its size.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0–0xFFFFF (1 Mbyte)</li> <li>● Maximum object size: 1 Mbyte.</li> <li>● Pointer size: 3 bytes.</li> </ul>                                                                                                                                                                                                                          |
| Example             | <code>__huge int x;</code>                                                                                                                                                                                                                                                                                                                                                                 |
| See also            | <i>Memory types</i> , page 66.                                                                                                                                                                                                                                                                                                                                                             |

## \_\_interrupt

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 337.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Description | <p>The <code>__interrupt</code> keyword specifies interrupt functions. To specify one or several interrupt vectors, use the <code>#pragma vector</code> 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.</p> <p>An interrupt function must have a <code>void</code> return type and cannot have any parameters.</p> <p>The header file <code>iodevice.h</code>, where <code>device</code> corresponds to the selected device, contains predefined names for the existing interrupt vectors.</p> <p> To make sure that the interrupt handler executes as fast as possible, you should compile it with <code>-Ohs</code>, or use <code>#pragma optimize=speed</code> if the module is compiled with another optimization goal.</p> |
| Example     | <pre>#pragma vector=0x14 __interrupt void my_interrupt_handler(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| See also    | <i>Interrupt functions</i> , page 79, and <i>.intvec</i> , page 444.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

**\_\_intrinsic**

Description The `__intrinsic` keyword is reserved for compiler internal use only.

**\_\_monitor**

Syntax See *Syntax for type attributes used on functions*, page 337.

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 80. For information about related intrinsic functions, see `__disable_interrupt`, page 382, `__enable_interrupt`, page 382, `__get_interrupt_state`, page 382, and `__set_interrupt_state`, page 384, respectively.

**\_\_near**

Syntax See *Syntax for type attributes used on data objects*, page 336.

Description The `__near` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in near memory. You can also use the `__near` attribute to create a pointer explicitly pointing to an object located in the near memory.

Storage information

- Address range: 0xF0000–0xFFFFF
- Maximum object size: 65535 bytes.
- Pointer size: 2 bytes.

Example 

```
__near int x;
```

See also *Memory types*, page 66.

## **\_\_near\_func**

|                     |                                                                                                                                                                                                                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on functions</i> , page 337.                                                                                                                                                                                                                                                                                |
| Description         | The <code>__near_func</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in <code>near_func</code> memory. You can also use the <code>__near_func</code> attribute to create a pointer explicitly pointing to an object located in <code>near_func</code> memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0-0xFFFF (64 Kbytes)</li> <li>● Maximum size: 65535 bytes.</li> <li>● Pointer size: 2 bytes</li> </ul>                                                                                                                                                                                     |
| Example             | <code>__near_func void myfunction(void);</code>                                                                                                                                                                                                                                                                                                    |
| See also            | <i>Code models and memory attributes for function storage</i> , page 77.                                                                                                                                                                                                                                                                           |

## **\_\_no\_alloc, \_\_no\_alloc16**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for object attributes</i> , page 339.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Description | <p>Use the <code>__no_alloc</code> or <code>__no_alloc16</code> object attribute on a constant to make the constant available in the executable file without occupying any space in the linked application.</p> <p>You cannot access the contents of such a constant from your application. You can take its address, which is an integer offset to the section of the constant. The type of the offset is <code>unsigned long</code> when <code>__no_alloc</code> is used, and <code>unsigned short</code> when <code>__no_alloc16</code> is used.</p> |
| Example     | <code>__no_alloc const struct MyData my_data @ "XXX" = {...};</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| See also    | <i>__no_alloc_str, __no_alloc_str16</i> , page 344.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

## **\_\_no\_alloc\_str, \_\_no\_alloc\_str16**

|        |                                                                                                                                  |
|--------|----------------------------------------------------------------------------------------------------------------------------------|
| Syntax | <code>__no_alloc_str(string_literal @ section)</code><br>and<br><code>__no_alloc_str16(string_literal @ section)</code><br>where |
|--------|----------------------------------------------------------------------------------------------------------------------------------|



|             |                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|             | <i>string_literal</i> | The string literal that you want to make available in the executable file.                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|             | <i>section</i>        | The name of the section to place the string literal in.                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description |                       | Use the <code>__no_alloc_str</code> or <code>__no_alloc_str16</code> operators to make string literals available in the executable file without occupying any space in the linked application.<br><br>The value of the expression is the offset of the string literal in the section. For <code>__no_alloc_str</code> , the type of the offset is unsigned long. For <code>__no_alloc_str16</code> , the type of the offset is unsigned short.                                                                     |
| Example     |                       | <pre>#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); }</pre> <p>Depending on your debugger and the runtime support, this could produce trace output on the host computer.</p> <p><b>Note:</b> There is no such runtime support in C-SPY, unless you use an external plugin module.</p> |
| See also    |                       | <code>__no_alloc</code> , <code>__no_alloc16</code> , page 344.                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## `__no_bit_access`

|             |                                                                                                                                                |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on data objects</i> , page 336.                                                                         |
| Description | For data objects declared <code>__no_bit_access</code> , the use of bit instructions is suppressed individual bits of the object are accessed. |

## **\_\_no\_init**

|             |                                                                                                                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for object attributes</i> , page 339.                                                                                                                                     |
| Description | Use the <code>__no_init</code> 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     | <pre>__no_init int myarray[10];</pre>                                                                                                                                                   |
| See also    | <i>Non-initialized variables</i> , page 235 and <i>do not initialize directive</i> , page 421.                                                                                          |

## **\_\_noreturn**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for object attributes</i> , page 339.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Description | <p>The <code>__noreturn</code> keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are <code>abort</code> and <code>exit</code>.</p> <p><b>Note:</b> At optimization levels Medium or High, the <code>__noreturn</code> keyword might cause incorrect call stack debug information at any point where it can be determined that the current function cannot return.</p> <p><b>Note:</b> The extended keyword <code>__noreturn</code> has the same meaning as the Standard C keyword <code>_Noreturn</code> or the macro <code>noreturn</code> (if <code>stdnoreturn.h</code> has been included) and as the Standard C++ attribute <code>[[noreturn]]</code>.</p> |
| Example     | <pre>__noreturn void terminate(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

## **\_\_no\_save**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 337.                                                                                                                                                                                                                                                                                                                                                                                    |
| Description | <p>Functions that are declared <code>__no_save</code> will not save and restore the workseg area that it has used.</p> <p>Functions normally save the used area of the <code>WRKSEG</code> segment on the stack. The <code>__no_save</code> keyword causes the function not to save this information when the <code>--workseg_area</code> option is used.</p> <p>This keyword has no effect unless the <code>--workseg_area</code> option is used.</p> |
| Example     | <pre>__no_save void myfunction(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                            |

## \_\_packed

### Syntax

See *Syntax for type attributes used on data objects*, page 336. An exception is when the keyword is used for modifying the structure type in a `struct` or `union` declarations, see below.

### Description

Use the `__packed` keyword to specify a data alignment of 1 for a data type. `__packed` can be used in two ways:

- When used before the `struct` or `union` keyword in a structure definition, the maximum alignment of each member in the structure is set to 1, eliminating the need for gaps between the members.  
You can also use the `__packed` keyword with structure declarations, but it is illegal to refer to a structure type defined without the `__packed` keyword using a structure declaration with the `__packed` keyword.
- When used in any other position, it follows the syntax rules for type attributes, and affects a type in its entirety. A type with the `__packed` type attribute is the same as the type attribute without the `__packed` type attribute, except that it has a data alignment of 1. Types that already have an alignment of 1 are not affected by the `__packed` type attribute.

A normal pointer can be implicitly converted to a pointer to `__packed`, but the reverse conversion requires a cast.

**Note:** Accessing data types at other alignments than their natural alignment can result in code that is significantly larger and slower.

Use either `__packed` or `#pragma pack` to relax the alignment restrictions for a type and the objects defined using that type. Mixing `__packed` and `#pragma pack` might lead to unexpected behavior.

### Example

```
/* No pad bytes in X: */
__packed struct X { char ch; int i; };
/* __packed is optional here: */
struct X * xp;

/* NOTE: no __packed: */
struct Y { char ch; int i; };
/* ERROR: Y not defined with __packed: */
__packed struct Y * yp ;

/* Member 'i' has alignment 1: */
struct Z { char ch; __packed int i; };
```

```
void Foo(struct X * xp)
{
 /* Error:"int __packed *" -> "int *" not allowed: */
 int * p1 = &xp->l;
 /* OK: */
 int __packed * p2 = &xp->i;
 /* OK, char not affected */
 char * p3 = &xp->ch;
}
```

See also *pack*, page 371.

## **\_\_root**

Syntax See *Syntax for object attributes*, page 339.

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 root symbols and how they are kept, see *Keeping symbols and sections*, page 109.

## **\_\_ro\_placement**

Syntax See *Syntax for object attributes*, page 339.

Unlike most object attributes, 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.

You can use the `__ro_placement` attribute with C++ objects if the compiler can optimize the C++ dynamic initialization of the data objects into static initialization. This is possible only for relatively simple constructors that have been defined in the header files of the relevant class definitions, so that they are visible to the compiler. If the compiler cannot find the constructor, or if the constructor is too complex, an error message will be issued (`Error[Go023]`) and the compilation will fail.

Example 

```
__ro_placement const volatile int x = 10;
```

## **\_\_saddr**

Syntax See *Syntax for type attributes used on data objects*, page 336.

Description The `__saddr` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in `saddr` memory.

You can also use the `__saddr` attribute to create a pointer explicitly pointing to an object located in the `saddr` memory.

Storage information

- Address range: 0xFFE20–0xFFF1F
- Maximum object size: 256 bytes.
- Pointer size: 2 bytes.

Example 

```
__saddr int x;
```

See also *Memory types*, page 66.

## **\_\_sfr**

Syntax See *Syntax for type attributes used on data objects*, page 336.

Description The `__sfr` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in the `sfr` area.

Storage information

- Address range: 0xFFF00–0xFFFFF
- Maximum object size: Typically, 256 bytes (depends on the device).
- Pointer size: n/a

Example 

```
__sfr int x;
```

See also *Memory types*, page 66.

## **\_\_v1\_call**

Syntax See *Syntax for type attributes used on functions*, page 337.

Description The `__v1_call` keyword makes the compiler use the calling convention of the IAR C/C++ Compiler for RL78 version 1.x for the function instead of the default calling convention. The V1 calling convention is required for backward compatibility with code written for the IAR C/C++ Compiler for RL78 version 1.x.

This keyword does not affect the bitfield allocation strategy.

**Note:** If you are migrating from version 1.x of the compiler, note that the stack is now cleaned by the calling function, not by the called function, regardless of which calling convention you use. Assembler functions that take stack parameters must be updated before you can use them.

Example 

```
__v1_call int func(int arg1, double arg2);
```

See also *Calling convention*, page 166.

## **\_\_v2\_call**

Syntax See *Syntax for type attributes used on functions*, page 337.

Description The `__v2_call` keyword makes the compiler use the V2 calling convention for the function. This is the default calling convention. The V2 calling convention complies with the RL78 ABI and is required when you link with code produced by tools from other vendors.

This keyword does not affect the bitfield allocation strategy.

Example 

```
__v2_call int func(int arg1, double arg2);
```

See also *Calling convention*, page 166.

## \_\_weak

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for object attributes</i> , page 339.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description | <p>Using the <code>__weak</code> object attribute on an external declaration of a symbol makes all references to that symbol in the module weak.</p> <p>Using the <code>__weak</code> object attribute on a public definition of a symbol makes that definition a weak definition.</p> <p>The linker will not include a module from a library solely to satisfy weak references to a symbol, nor will the lack of a definition for a weak reference result in an error. If no definition is included, the address of the object will be zero.</p> <p>When linking, a symbol can have any number of weak definitions, and at most one non-weak definition. If the symbol is needed, and there is a non-weak definition, this definition will be used. If there is no non-weak definition, one of the weak definitions will be used.</p> |
| Example     | <pre>extern __weak int foo; /* A weak reference. */  __weak void bar(void) /* A weak definition. */ {     /* Increment foo if it was included. */     if (&amp;foo != 0)         ++foo; }</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

---

## Supported GCC attributes

In extended language mode, the IAR C/C++ Compiler also supports a limited selection of GCC-style attributes. Use the `__attribute__((attribute-list))` syntax for these attributes.

The following attributes are supported in part or in whole. For more information, see the GCC documentation.

- `alias`
- `aligned`
- `always_inline`
- `constructor`
- `deprecated`
- `noinline`

- `noreturn`
- `packed`
- `pcs` (for IAR type attributes used on functions)
- `section`
- `target` (for IAR object attributes used on functions)
- `transparent_union`
- `unused`
- `used`
- `volatile`
- `weak`



# 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:

| <b>Pragma directive</b>                  | <b>Description</b>                                                                     |
|------------------------------------------|----------------------------------------------------------------------------------------|
| <code>bank</code>                        | Specifies the register bank number of the immediately following interrupt function.    |
| <code>basic_template_matching</code>     | Makes a template function fully memory-attribute aware.                                |
| <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 section.                                          |
| <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 section.                                                   |
| <code>default_function_attributes</code> | Sets default type and object attributes for declarations and definitions of functions. |

*Table 32: Pragma directives summary*

| <b>Pragma directive</b>                  | <b>Description</b>                                                                                          |
|------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>default_variable_attributes</code> | Sets default type and object attributes for declarations and definitions of variables.                      |
| <code>deprecated</code>                  | Marks an entity as deprecated.                                                                              |
| <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>function_category</code>           | Declares function categories for stack usage analysis.                                                      |
| <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 sections. |
| <code>message</code>                     | Prints a message.                                                                                           |
| <code>no_stack_protect</code>            | Disables stack protection for the following function.                                                       |
| <code>no_workseg</code>                  | Specifies that the immediately following function does not use the workseg area.                            |
| <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>                     | Declares a section name to be used by intrinsic functions.                                                  |

---

*Table 32: Pragma directives summary (Continued)*

| Pragma directive                   | Description                                                                                 |
|------------------------------------|---------------------------------------------------------------------------------------------|
| <code>segment</code>               | This directive is an alias for <code>#pragma section</code> .                               |
| <code>stack_protect</code>         | Forces stack protection for the function that follows.                                      |
| <code>STDC CX_LIMITED_RANGE</code> | Specifies whether the compiler can use normal complex mathematical formulas or not.         |
| <code>STDC FENV_ACCESS</code>      | Specifies whether your source code accesses the floating-point environment or not.          |
| <code>STDC FP_CONTRACT</code>      | Specifies whether the compiler is allowed to contract floating-point expressions or not.    |
| <code>type_attribute</code>        | Adds type attributes to a declaration or to definitions.                                    |
| <code>unroll</code>                | Unrolls loops.                                                                              |
| <code>vector</code>                | Specifies the vector of an interrupt or trap function.                                      |
| <code>weak</code>                  | Makes a definition a weak definition, or creates a weak alias for a function or a variable. |

Table 32: Pragma directives summary (Continued)

**Note:** For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 531.

## Descriptions of pragma directives

This section gives detailed information about each pragma directive.

### bank

|             |                                                                                                                                                                                                                                                                                   |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma bank=bank_number</code>                                                                                                                                                                                                                                             |
| Parameters  | <p><i>bank_number</i>      A constant integer expression that denotes one of the register banks 0–3.</p>                                                                                                                                                                          |
| Description | Use this pragma directive to declare an interrupt function that switches to another register bank before starting the execution of the function. The <code>#pragma bank</code> directive controls the register bank number (0–3) of the immediately following interrupt function. |

## 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 195. |
| Example     | <pre>#pragma basic_template_matching template&lt;typename T&gt; void fun(T *);  fun((int __near *) 0); /* Template parameter T becomes                        int __near */</pre>                                                                                                                           |

## bitfields

|            |                                                                                                      |                                                                                                                                                                                                                                                                                               |
|------------|------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>#pragma bitfields=disjoint_types joined_types reversed_disjoint_types reversed default}</code> |                                                                                                                                                                                                                                                                                               |
| Parameters | <code>disjoint_types</code>                                                                          | Bitfield members are placed from the least significant bit to the most significant bit in the container type. Storage containers of bitfields with different base types will not overlap.                                                                                                     |
|            | <code>joined_types</code>                                                                            | Bitfield members are placed depending on the byte order. Storage containers of bitfields will overlap other structure members. For more information, see <i>Bitfields</i> , page 324. This parameter is not available if you use the V1 calling convention as the default calling convention. |
|            | <code>reversed_disjoint_types</code>                                                                 | Bitfield members are placed from the most significant bit to the least significant bit in the container type. Storage containers of bitfields with different base types will not overlap.                                                                                                     |
|            | <code>reversed</code>                                                                                | This is an alias for <code>reversed_disjoint_types</code> .                                                                                                                                                                                                                                   |

`default` Restores the default layout of bitfield members. If you use the V2 calling convention as the default calling convention, the default behavior for the compiler is `joined_types`. If you use the V1 calling convention as the default calling convention, the default behavior for the compiler is `disjoint_types`.

**Description** Use this pragma directive to control the layout of bitfield members.

**Example**

```
#pragma bitfields=disjoint_types
/* Structure that uses disjoint bitfield types. */
struct S
{
 unsigned char error : 1;
 unsigned char size : 4;
 unsigned short code : 10;
};
#pragma bitfields=default /* Restores to default setting. */
```

**See also** *Calling convention*, page 166 and *Bitfields*, page 324.

## calls

**Syntax** `#pragma calls=arg[, arg...]`

**Parameters** `arg` can be one of these:

*function* A declared function

*category* A string that represents the name of a function category

**Description** Use this pragma directive to specify all functions that can be indirectly called in the following statement. This information can be used for stack usage analysis in the linker. You can specify individual functions or function categories. Specifying a category is equivalent to specifying all included functions in that category.

**Example**

```
void Fun1(), Fun2();

void Caller(void (*fp)(void))
{
#pragma calls = Fun1, Fun2, "Cat1"
 (*fp)(); // Can call Fun1, Fun2, and all
 // functions in category "Cat1"
}

See also function_category, page 365 and Stack usage analysis, page 97.

```

## call\_graph\_root

**Syntax** `#pragma call_graph_root [=category]`

**Parameters**

|                 |                                                               |
|-----------------|---------------------------------------------------------------|
| <i>category</i> | 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 97.

## constseg

**Syntax** `#pragma constseg=[__memoryattribute ]{SECTION_NAME|default}`

**Parameters**

|                          |                                                                                                                            |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <i>__memoryattribute</i> | An optional memory attribute denoting in what memory the section will be placed; if not specified, default memory is used. |
| <i>SECTION_NAME</i>      | A user-defined section name; cannot be a section name predefined for use by the compiler and linker.                       |
| default                  | Uses the default section for constants.                                                                                    |

**Description** Use this pragma directive to place constant variables in a named section. The section name cannot be a section name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

**Example**

```
#pragma constseg=__far MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

## data\_alignment

**Syntax** `#pragma data_alignment=expression`

**Parameters**

*expression*                      A constant which must be a power of two (1, 2, 4, etc.).

**Description** Use this pragma directive to give the immediately following variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

**Note:** Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can therefore be used for creating situations where the size is not a multiple of the alignment.

**Note:** To comply with the ISO C11 standard and later, it is recommended to use the alignment specifier `_Alignas` for C code. To comply with the C++11 standard and later, it is recommended to use the alignment specifier `alignas` for C++ code.

## dataseg

**Syntax** `#pragma dataseg=[__memoryattribute]{SECTION_NAME|default}`

**Parameters**

*\_\_memoryattribute*              An optional memory attribute denoting in what memory the section will be placed; if not specified, default memory is used.

|             |                     |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|             | <i>SECTION_NAME</i> | A user-defined section name; cannot be a section name predefined for use by the compiler and linker.                                                                                                                                                                                                                                                                                                                                 |
|             | default             | Uses the default section.                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description |                     | Use this pragma directive to place variables in a named section. The section name cannot be a section 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

```
#pragma dataseg=__far MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

## default\_function\_attributes

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma default_function_attributes=[ attribute...]</code><br>where <i>attribute</i> can be:<br><br><i>type_attribute</i><br><i>object_attribute</i><br><i>@ section_name</i>                                                                                                                                                                                                                                                                                                                            |
| Parameters  | <i>type_attribute</i> See <i>Type attributes</i> , page 335.<br><i>object_attribute</i> See <i>Object attributes</i> , page 338.<br><i>@ section_name</i> See <i>Data and function placement in sections</i> , page 224.                                                                                                                                                                                                                                                                                       |
| Description | Use this pragma directive to set default section 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.<br><br>Specifying a <code>default_function_attributes</code> 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 section MYSEC" */
#pragma default_function_attributes = @ "MYSEC"
int fun1(int x) { return x + 1; }
int fun2(int x) { return x - 1; }
/* Stop placing functions into MYSEC */
#pragma default_function_attributes =
```

has the same effect as:

```
int fun1(int x) @ "MYSEC" { return x + 1; }
int fun2(int x) @ "MYSEC" { return x - 1; }
```

**See also**

*location*, page 368.

*object\_attribute*, page 369.

*type\_attribute*, page 377.

**default\_variable\_attributes****Syntax**

```
#pragma default_variable_attributes=[attribute...]
```

where *attribute* can be:

```
type_attribute
object_attribute
@ section_name
```

**Parameters**

*type\_attribute*            See *Type attributes*, page 335.

*object\_attributes*       See *Object attributes*, page 338.

@ *section\_name*           See *Data and function placement in sections*, page 224.

**Description**

Use this pragma directive to set default section 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 directive with no attributes restores the initial state of no such defaults being applied to variables with static storage duration.

**Example**

```
/* Place following variables in section MYSEC */
#pragma default_variable_attributes = @ "MYSEC"
int var1 = 42;
int var2 = 17;
/* Stop placing variables into MYSEC */
#pragma default_variable_attributes =
```

has the same effect as:

```
int var1 @ "MYSEC" = 42;
int var2 @ "MYSEC" = 17;
```

**See also**

*location*, page 368.

*object\_attribute*, page 369.

*type\_attribute*, page 377.

## deprecated

**Syntax**

```
#pragma deprecated=entity
```

**Description**

If you place this pragma directive immediately before the declaration of a type, variable, function, field, or constant, any use of that type, variable, function, field, or constant will result in a warning.

The `deprecated` pragma directive has the same effect as the C++ attribute `[[deprecated]]`, but is available in C as well.

**Example**

```
#pragma deprecated
typedef int * intp_t; // typedef intp_t is deprecated

#pragma deprecated
extern int fun(void); // function fun is deprecated

#pragma deprecated
struct xx { // struct xx is deprecated
 int x;
};
```

```

struct yy {
#pragma deprecated
 int y; // field y is deprecated
};

intp_t fun(void) // Warning here
{
 struct xx ax; // Warning here
 struct yy ay;
 fun(); // Warning here
 return ay.y; // Warning here
}

```

See also [Annex K \(\*Bounds-checking interfaces\*\)](#) of the C standard.

## 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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also [Diagnostics](#), page 246.

## 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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also *Diagnostics*, page 246.

## **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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also *Diagnostics*, page 246.

## **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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also *Diagnostics*, page 246.

## **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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also *Diagnostics*, page 246.

## 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.

## function\_category

Syntax `#pragma function_category=category[, category...]`

Parameters *category*                    A string that represents the name of a function category.

Description Use this pragma directive to specify one or more function categories that the immediately following function belongs to. When used together with `#pragma calls`, the `function_category` directive specifies the destination for indirect calls for stack usage analysis purposes.

Example `#pragma function_category="Cat1"`

See also *calls*, page 357 and *Stack usage analysis*, page 97.

## include\_alias

|             |                                                                                                                                                                                                                                                                                                                                                                      |                                                                  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| Syntax      | <pre>#pragma include_alias ("orig_header" , "subst_header") #pragma include_alias (&lt;orig_header&gt; , &lt;subst_header&gt;)</pre>                                                                                                                                                                                                                                 |                                                                  |
| Parameters  | <i>orig_header</i>                                                                                                                                                                                                                                                                                                                                                   | The name of a header file for which you want to create an alias. |
|             | <i>subst_header</i>                                                                                                                                                                                                                                                                                                                                                  | The alias for the original header file.                          |
| Description | <p>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.</p> <p>This pragma directive must appear before the corresponding #include directives and <i>subst_header</i> must match its corresponding #include directive exactly.</p> |                                                                  |
| Example     | <pre>#pragma include_alias (&lt;stdio.h&gt; , &lt;C:\MyHeaders\stdio.h&gt;) #include &lt;stdio.h&gt;</pre> <p>This example will substitute the relative file <code>stdio.h</code> with a counterpart located according to the specified path.</p>                                                                                                                    |                                                                  |
| See also    | <i>Include file search procedure</i> , page 241.                                                                                                                                                                                                                                                                                                                     |                                                                  |

## inline

|             |                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| Syntax      | <pre>#pragma inline[=forced =never]</pre>                                                                                                                                                                                                                                                                                                                                                            |                                                                                          |
| 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 | <p>Use <code>#pragma inline</code> to advise the compiler that the function defined immediately after the directive should be inlined according to C++ inline semantics.</p> <p>Specifying <code>#pragma inline=forced</code> 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.</p> |                                                                                          |

Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` will inline the function or result in an error due to recursion etc.

See also *Inlining functions*, page 83.

## 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.<br><br>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 At the top of a file that needs to be compiled with IAR Systems extensions enabled:

```
#pragma language=extended
/* The rest of the file. */
```

Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled, but where the state before the sequence cannot be assumed to be the same as that specified by the compiler options in use:

```
#pragma language=save
#pragma language=extended
/* Part of source code. */
#pragma language=restore
```

See also `-e`, page 273 and `--strict`, page 304.

## location

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma location={<i>address</i> <i>NAME</i>}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Parameters  | <p><i>address</i>                      The absolute address of the global or static variable for which you want an absolute location.</p> <p><i>NAME</i>                              A user-defined section name—cannot be a section name predefined for use by the compiler and linker.</p>                                                                                                                                                                                                                                        |
| Description | <p>Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. Alternatively, the directive can take a string specifying a section for placing either a variable or a function whose declaration follows the pragma directive. Do not place variables that would normally be in different sections—for example, variables declared as <code>__no_init</code> and variables declared as <code>const</code>—in the same named section.</p> |
| Example     | <pre>#pragma location=0xFFFF20 __no_init volatile char __sfr PORT1; /* PORT1 is located at address 0xFFFF20 */  #pragma segment="FLASH" #pragma location="FLASH" __no_init char PORT2; /* PORT2 is located in segment FLASH */  /* A better way is to use a corresponding mechanism */ #define FLASH _Pragma("location=\"FLASH\"")  FLASH __no_init int i; /* i is placed in the FLASH segment */</pre>                                                                                                                              |
| See also    | <p><i>Controlling data and function placement in memory</i>, page 222 and <i>Declare and place your own sections</i>, page 108.</p>                                                                                                                                                                                                                                                                                                                                                                                                  |

## message

|            |                                                                                                               |
|------------|---------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>#pragma message(<i>message</i>)</code>                                                                  |
| Parameters | <p><i>message</i>                      The message that you want to direct to the standard output stream.</p> |



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\_stack\_protect

Syntax `#pragma no_stack_protect`

Description Use this pragma directive to disable stack protection for the defined function that follows.

This pragma directive only has effect if the compiler option `--stack_protection` has been used.

See also *Stack protection*, page 85.

## no\_workseg

Syntax `#pragma no_workseg`

Description Use this pragma directive in front of a function declaration to prevent the function from using the register variable work area, created with the `--workseg_area` compiler option.

See also *Short address working area*, page 74.

## object\_attribute

Syntax `#pragma object_attribute=object_attribute[ object_attribute...]`

Parameters For information about object attributes that can be used with this pragma directive, see *Object attributes*, page 338.

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 335.                                                   |

## optimize

|                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |             |                                                                                                                                                                                                                                                                                                                                                 |              |                                                                                                                |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|----------------------------------------------------------------------------------------------------------------|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax         | <code>#pragma optimize=[<i>goal</i>] [<i>level</i>] [<i>disable</i>]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |             |                                                                                                                                                                                                                                                                                                                                                 |              |                                                                                                                |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Parameters     | <table> <tr> <td style="vertical-align: top;"><i>goal</i></td> <td> <p>Choose between:</p> <ul style="list-style-type: none"> <li><i>size</i>, optimizes for size</li> <li><i>balanced</i>, optimizes balanced between speed and size</li> <li><i>speed</i>, optimizes for speed.</li> <li><i>no_size_constraints</i>, optimizes for speed, but relaxes the normal restrictions for code size expansion.</li> </ul> </td> </tr> <tr> <td style="vertical-align: top;"><i>level</i></td> <td>Specifies the level of optimization—choose between <i>none</i>, <i>low</i>, <i>medium</i>, or <i>high</i>.</td> </tr> <tr> <td style="vertical-align: top;"><i>disable</i></td> <td> <p>Disables one or several optimizations (separated by spaces). Choose between:</p> <ul style="list-style-type: none"> <li><i>no_code_motion</i>, disables code motion</li> <li><i>no_cse</i>, disables common subexpression elimination</li> <li><i>no_inline</i>, disables function inlining</li> <li><i>no_relaxed_fp</i>, disables the language relaxation that optimizes floating-point expressions more aggressively</li> <li><i>no_tbaa</i>, disables type-based alias analysis</li> <li><i>no_unroll</i>, disables loop unrolling</li> </ul> </td> </tr> </table> | <i>goal</i> | <p>Choose between:</p> <ul style="list-style-type: none"> <li><i>size</i>, optimizes for size</li> <li><i>balanced</i>, optimizes balanced between speed and size</li> <li><i>speed</i>, optimizes for speed.</li> <li><i>no_size_constraints</i>, optimizes for speed, but relaxes the normal restrictions for code size expansion.</li> </ul> | <i>level</i> | Specifies the level of optimization—choose between <i>none</i> , <i>low</i> , <i>medium</i> , or <i>high</i> . | <i>disable</i> | <p>Disables one or several optimizations (separated by spaces). Choose between:</p> <ul style="list-style-type: none"> <li><i>no_code_motion</i>, disables code motion</li> <li><i>no_cse</i>, disables common subexpression elimination</li> <li><i>no_inline</i>, disables function inlining</li> <li><i>no_relaxed_fp</i>, disables the language relaxation that optimizes floating-point expressions more aggressively</li> <li><i>no_tbaa</i>, disables type-based alias analysis</li> <li><i>no_unroll</i>, disables loop unrolling</li> </ul> |
| <i>goal</i>    | <p>Choose between:</p> <ul style="list-style-type: none"> <li><i>size</i>, optimizes for size</li> <li><i>balanced</i>, optimizes balanced between speed and size</li> <li><i>speed</i>, optimizes for speed.</li> <li><i>no_size_constraints</i>, optimizes for speed, but relaxes the normal restrictions for code size expansion.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |             |                                                                                                                                                                                                                                                                                                                                                 |              |                                                                                                                |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>level</i>   | Specifies the level of optimization—choose between <i>none</i> , <i>low</i> , <i>medium</i> , or <i>high</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |             |                                                                                                                                                                                                                                                                                                                                                 |              |                                                                                                                |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>disable</i> | <p>Disables one or several optimizations (separated by spaces). Choose between:</p> <ul style="list-style-type: none"> <li><i>no_code_motion</i>, disables code motion</li> <li><i>no_cse</i>, disables common subexpression elimination</li> <li><i>no_inline</i>, disables function inlining</li> <li><i>no_relaxed_fp</i>, disables the language relaxation that optimizes floating-point expressions more aggressively</li> <li><i>no_tbaa</i>, disables type-based alias analysis</li> <li><i>no_unroll</i>, disables loop unrolling</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |             |                                                                                                                                                                                                                                                                                                                                                 |              |                                                                                                                |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 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 228.

## pack

#### Syntax

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop} [, name] [, n])
```

#### Parameters

|                   |                                                                      |
|-------------------|----------------------------------------------------------------------|
| <i>n</i>          | Sets an optional structure alignment—one of: 1, 2, 4, 8, or 16       |
| Empty list        | Restores the structure alignment to default                          |
| <code>push</code> | Sets a temporary structure alignment                                 |
| <code>pop</code>  | Restores the structure alignment from a temporarily pushed alignment |
| <i>name</i>       | An optional pushed or popped alignment label                         |

#### Description

Use this pragma directive to specify the maximum alignment of `struct` and `union` members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or the end of the compilation unit.

**Note:** This can result in significantly larger and slower code when accessing members of the structure.

Use either `__packed` or `#pragma pack` to relax the alignment restrictions for a type and the objects defined using that type. Mixing `__packed` and `#pragma pack` might lead to unexpected behavior.

See also

*Structure types*, page 330 and *\_\_packed*, page 347.

## `__printf_args`

Syntax

```
#pragma __printf_args
```

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier, for example `%d`, is syntactically correct.

You cannot use this pragma directive on functions that are members of an overload set with more than one member.

Example

```
#pragma __printf_args
int printf(char const *,...);

void PrintNumbers(unsigned short x)
{
 printf("%d", x); /* Compiler checks that x is an integer */
}
```

## `public_equ`

Syntax

```
#pragma public_equ=symbol, value
```

Parameters

|               |                                                                          |
|---------------|--------------------------------------------------------------------------|
| <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

```
#pragma public_equ="MY_SYMBOL", 0x123456
```

See also `--public_equ`, page 297.

## required

|             |                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma required=symbol</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 section 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 <code>copyright</code> string is not used by the application, it will still be included by the linker and available in the output.</p>                                                                                                               |

## rtmodel

|             |                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma rtmodel="key", "value"</code>                                                                                                                                                                                                                                                                                              |
| Parameters  | <p><code>"key"</code>                      A text string that specifies the runtime model attribute.</p> <p><code>"value"</code>                    A text string that specifies the value of the runtime model attribute. Using the special value <code>*</code> is equivalent to not defining the attribute at all.</p>                |
| Description | <p>Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.</p> <p>This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same</p> |

value for the corresponding key, or the special value \*. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

**Note:** The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C", "ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

## \_\_scanf\_args

Syntax

```
#pragma __scanf_args
```

Description

Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier, for example %d, is syntactically correct.

You cannot use this pragma directive on functions that are members of an overload set with more than one member.

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;
}
```

## section

Syntax

```
#pragma section="NAME" [__memoryattribute]
alias
#pragma segment="NAME" [__memoryattribute]
```

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Parameters  | <p><i>NAME</i> The name of the section.</p> <p><i>__memoryattribute</i> An optional memory attribute identifying the memory the section will be placed in; if not specified, default memory is used.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Description | <p>Use this pragma directive to define a section name that can be used by the section operators <code>__section_begin</code>, <code>__section_end</code>, and <code>__section_size</code>. All section declarations for a specific section must have the same memory type attribute and alignment.</p> <p>The <i>__memoryattribute</i> parameter only relevant when used together with the section operators <code>__section_begin</code>, <code>__section_end</code>, and <code>__section_size</code>.</p> <p>If an optional memory attribute is used, the return type of the section operators <code>__section_begin</code> and <code>__section_end</code> is:</p> <pre>void __memoryattribute *.</pre> <p><b>Note:</b> To place variables or functions in a specific section, use the <code>#pragma location</code> directive or the <code>@</code> operator.</p> |
| Example     | <pre>#pragma section="MYNEAR" __near</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| See also    | <i>Dedicated section operators</i> , page 186 and the chapter <i>Linking your application</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

## stack\_protect

|             |                                                                                            |
|-------------|--------------------------------------------------------------------------------------------|
| Syntax      | <pre>#pragma stack_protect</pre>                                                           |
| Description | Use this pragma directive to force stack protection for the defined function that follows. |
| See also    | <i>Stack protection</i> , page 85.                                                         |

## STDC\_CX\_LIMITED\_RANGE

|            |                                                                                                                         |
|------------|-------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <pre>#pragma STDC CX_LIMITED_RANGE {ON OFF DEFAULT}</pre>                                                               |
| Parameters | <p>ON Normal complex mathematic formulas can be used.</p> <p>OFF Normal complex mathematic formulas cannot be used.</p> |

|             |         |                                                                                                                                                                                                                                                                                   |
|-------------|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|             | 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.<br><br><b>Note:</b> 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.<br><br><b>Note:</b> 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.<br><br><b>Note:</b> This directive is required by Standard C. |

## STDC FP\_CONTRACT

|             |                                           |                                                                                                                                                               |
|-------------|-------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | #pragma STDC FP_CONTRACT {ON OFF DEFAULT} |                                                                                                                                                               |
| Parameters  | ON                                        | The compiler is allowed to contract floating-point expressions.                                                                                               |
|             | OFF                                       | The compiler is not allowed to contract floating-point expressions.<br><br><b>Note:</b> This argument is not supported by the compiler.                       |
|             | DEFAULT                                   | Sets the default behavior, that is ON.                                                                                                                        |
| Description |                                           | Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C. |



Example `#pragma STDC FP_CONTRACT=ON`

## type\_attribute

Syntax `#pragma type_attribute=type_attr[ type_attr...]`

Parameters For information about type attributes that can be used with this pragma directive, see *Type attributes*, page 335.

Description Use this pragma directive to specify IAR-specific *type attributes*, which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example In this example, an `int` object with the memory attribute `__near` is defined:

```
#pragma type_attribute=__near
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__near int x;
```

See also The chapter *Extended keywords*.

## unroll

Syntax `#pragma unroll=n`

Parameters `n` The number of loop bodies in the unrolled loop, a constant integer. `#pragma unroll = 1` will prevent the unrolling of a loop.

Description Use this pragma directive to specify that the loop following immediately after the directive should be unrolled and that the unrolled loop should have `n` copies of the loop body. The pragma directive can only be placed immediately before a `for`, `do`, or `while` loop, whose number of iterations can be determined at compile time.

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.

The `#pragma unroll` 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.

Example

```
#pragma unroll=4
for (i = 0; i < 64; ++i)
{
 foo(i * k, (i + 1) * k);
}
```

See also *Loop unrolling*, page 229

## vector

Syntax `#pragma vector=vector1[, vector2, vector3, ...]`

Parameters *vectorN* The vector number(s) of an interrupt function.

Description Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example

```
#pragma vector=0x14
__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. The `__weak` attribute can also be used for this purpose.

- 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`.

**See also**

`__weak`, page 351.



# Intrinsic functions

- Summary of intrinsic functions
- Descriptions of intrinsic functions

---

## Summary of intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

To 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:

| <b>Intrinsic function</b>          | <b>Description</b>                                |
|------------------------------------|---------------------------------------------------|
| <code>__break</code>               | Inserts a BRK instruction                         |
| <code>__disable_interrupt</code>   | Disables interrupts                               |
| <code>__enable_interrupt</code>    | Enables interrupts                                |
| <code>__get_interrupt_level</code> | Returns the interrupt level                       |
| <code>__get_interrupt_state</code> | Returns the interrupt state                       |
| <code>__halt</code>                | Inserts a halt/no-operation instruction pair      |
| <code>__mach</code>                | Inserts a MACH instruction (S3 core)              |
| <code>__machu</code>               | Inserts a MACHU instruction (S3 core)             |
| <code>__no_operation</code>        | Inserts a NOP instruction                         |
| <code>__rollb</code>               | Rotates a <code>char</code> value one step left   |
| <code>__rollw</code>               | Rotates a <code>short</code> value one step left  |
| <code>__ror1b</code>               | Rotates a <code>char</code> value one step right  |
| <code>__ror1w</code>               | Rotates a <code>short</code> value one step right |
| <code>__set_interrupt_level</code> | Sets the interrupt level                          |
| <code>__set_interrupt_state</code> | Restores the interrupt state                      |
| <code>__stop</code>                | Inserts a STOP instruction                        |

*Table 33: Intrinsic functions summary*

---

## Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

### **\_\_break**

Syntax `void __break(void);`

Description Inserts a `BRK` instruction.

### **\_\_disable\_interrupt**

Syntax `void __disable_interrupt(void);`

Description Disables interrupts by inserting the `DI` instruction.

### **\_\_enable\_interrupt**

Syntax `void __enable_interrupt(void);`

Description Enables interrupts by inserting the `EI` instruction.

### **\_\_get\_interrupt\_level**

Syntax `__ilevel_t __get_interrupt_level(void);`

Description Returns the current interrupt level. The return type `__ilevel_t` has this definition:

```
typedef unsigned char __ilevel_t;
```

The return value of `__get_interrupt_level` can be used as an argument to the `__set_interrupt_level` intrinsic function.

### **\_\_get\_interrupt\_state**

Syntax `__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** 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`.

## **\_\_halt**

**Syntax** `void __halt(void);`

**Description** Inserts a halt/no-operation instruction pair.

## **\_\_mach**

**Syntax** `void __mach(signed short, signed short);`

**Description** Inserts a MACH instruction. This intrinsic function can only be used in code compiled for the S3 core.

## **\_\_machu**

**Syntax** `void __machu(unsigned short, unsigned short);`

**Description** Inserts a MACHU instruction. This intrinsic function can only be used in code compiled for the S3 core.

## **\_\_no\_operation**

**Syntax** `void __no_operation(void);`

**Description** Inserts a instruction.

## **\_\_rollb**

**Syntax** `unsigned char __rollb(unsigned char);`

**Description** Rotates the `char` argument one step left.

### **\_\_rollw**

Syntax `unsigned short __rollw(unsigned short);`

Description Rotates the `short` argument one step left.

### **\_\_rorlb**

Syntax `unsigned char __rorlb(unsigned char);`

Description Rotates the `char` argument one step right.

### **\_\_rorlw**

Syntax `unsigned short __rorlw(unsigned short);`

Description Rotates the `short` argument one step right.

### **\_\_set\_interrupt\_level**

Syntax `void __set_interrupt_level(__ilevel_t);`

Description Sets the interrupt level. For information about the `__ilevel_t` type, see [\\_\\_get\\_interrupt\\_level](#), page 382.

### **\_\_set\_interrupt\_state**

Syntax `void __set_interrupt_state(__istate_t);`

Description Restores the interrupt state to a value previously returned by the `__get_interrupt_state` function.

For information about the `__istate_t` type, see [\\_\\_get\\_interrupt\\_state](#), page 382.

### **\_\_stop**

Syntax `void __stop(void);`

Description Inserts a `STOP` instruction.



# 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 RL78 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 386.
- 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 259.
- 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 391.
- Preprocessor output

Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 282.

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:** Backslashes can also be used—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.

**Note:** To list the predefined preprocessor symbols, use the compiler option `--predef_macros`. See *--predef\_macros*, page 281.

### `__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 388, and `--no_path_in_file_macros`, page 276.

### `__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.

### `__CALLING_CONVENTION__`

**Description** Identifies the default calling convention in use. The value reflects the setting of the `--calling_convention` option and is defined to `__CC_V1__` or `__CC_V2__`. These symbolic names can be used when testing the `__CALLING_CONVENTION__` symbol.

**See also** *Calling convention*, page 166 and `--calling_convention`, page 256.

### `__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_NEAR__` or `__CODE_MODEL_FAR__`. 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 `__S1__`, `__S2__`, or `__S3__`. These symbolic names can be used when testing the `__CORE__` symbol.

**Note:** For backward compatibility, this symbol can also be tested against the values `__RL78_0__`, `__RL78_1__`, and `__RL78_2__`.

## **\_\_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 201402L. 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_NEAR__`, `__DATA_MODEL_FAR__`, or `__DATA_MODEL_HUGE__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.

## **\_\_DATE\_\_**

Description

A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example, "Oct 30 2018".

This symbol is required by Standard C.

## **\_\_EXCEPTIONS\_\_**

Description

A symbol that is defined when exceptions are supported in C++.

## **\_\_FAR\_RUNTIME\_ATTRIBUTE\_\_**

Description

A symbol that is defined to `__far_func` when the `--generate_far_runtime_library_calls` option is used and to `__near_func` otherwise.

See also `--generate_far_runtime_library_calls`, page 269.

## **\_\_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 386, and `--no_path_in_file_macros`, page 276.

## **\_\_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 267 and `__PRETTY_FUNCTION__`, page 389.

## **\_\_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 267 and `__PRETTY_FUNCTION__`, page 389.

## **\_\_IAR\_SYSTEMS\_ICC\_\_**

Description An integer that identifies the IAR compiler platform. The current value is 9—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.

## **\_\_ICCRL78\_\_**

Description An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for RL78.

**\_\_LINE\_\_**

Description An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

**\_\_LITTLE\_ENDIAN\_\_**

Description An integer that reflects the byte order and is defined to 1 (little-endian).

**\_\_PRETTY\_FUNCTION\_\_**

Description A predefined string identifier that is initialized with the function name, including parameter types and return type, of the function in which the symbol is used, for example, "void func(char)". This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

See also `-e`, page 267 and `__func__`, page 388.

**\_\_RTTI\_\_**

Description A symbol that is defined when runtime type information (RTTI) is supported in C++.

**\_\_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\_LIB\_EXT1\_\_**

Description An integer that is set to 201112L and that signals that Annex K, *Bounds-checking interfaces*, of the C standard is supported.

See also `__STDC_WANT_LIB_EXT1__`, page 392.

## **\_\_STDC\_NO\_ATOMICS\_\_**

Description Set to 1 if the compiler does not support atomic types nor `stdatomic.h`.

## **\_\_STDC\_NO\_THREADS\_\_**

Description Set to 1 to indicate that the implementation does not support threads.

## **\_\_STDC\_NO\_VLA\_\_**

Description Set to 1 to indicate that C variable length arrays, VLAs, are not enabled.

See also `--vla`, page 289.

## **\_\_STDC\_UTF16\_\_**

Description Set to 1 to indicate that the values of type `char16_t` are UTF-16 encoded.

## **\_\_STDC\_UTF32\_\_**

Description Set to 1 to indicate that the values of type `char32_t` are UTF-32 encoded.

## **\_\_STDC\_VERSION\_\_**

Description An integer that identifies the version of the C standard in use. The symbol expands to 201710L, unless the `--c89` compiler option is used, in which case the symbol expands to 199409L.

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:** 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 `__iar_ReportAssert`, page 149.

## **`__STDC_WANT_LIB_EXT1__`**

**Description** If this symbol is defined to 1 prior to any inclusions of system header files, it will enable the use of functions from Annex K, *Bounds-checking interfaces*, of the C standard.

**See also** *Bounds checking functionality*, page 131 and *C bounds-checking interface*, page 402.

## **`#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

For detailed reference information about the library functions, see the online help system.

---

## C/C++ standard library overview

**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.

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

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*.

## 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 set up a runtime library, see *Setting up the runtime environment*, page 125. The linker will include only those routines that are required—directly or indirectly—by your application.

For information about how you can override library modules with your own versions, see *Overriding library modules*, page 128.

## 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 `--redirect` 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`, etc. and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mblen`, `mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `perror`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `scanf`, `sscanf`, `getchar`, `getwchar`, `putchar`, and `putwchar`. 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.

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

These topics are covered:

- *Briefly about the DLIB runtime environment*, page 395
- *C header files*, page 396
- *C++ header files*, page 397
- *Library functions as intrinsic functions*, page 401
- *Not supported C/C++ functionality*, page 401
- *Atomic operations*, page 401
- *Added C functionality*, page 401
- *Non-standard implementations*, page 404
- *Symbols used internally by the library*, page 404

### BRIEFLY ABOUT THE DLIB RUNTIME ENVIRONMENT

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 more information, see the chapter *Implementation-defined behavior for Standard C*.
- 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*.

- Runtime support libraries, for example, low-level floating-point routines.
- Intrinsic functions, allowing low-level use of RL78 features. For more information, see the chapter *Intrinsic functions*.

In addition, the DLIB runtime environment includes some added C functionality, see *Added C functionality*, page 401.

## C HEADER FILES

This section lists the C header files specific to the DLIB runtime environment. Header files may additionally contain target-specific definitions, which 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>    | Alternative spellings                                                         |
| <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>stdalign.h</code>  | Handling alignment on data objects                                            |
| <code>stdarg.h</code>    | Accessing a varying number of arguments                                       |
| <code>stdatomic.h</code> | Adding support for atomic operations.<br>This functionality is not supported. |
| <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                                            |

Table 34: Traditional Standard C header files—DLIB

| Header file                | Usage                                                                                    |
|----------------------------|------------------------------------------------------------------------------------------|
| <code>stdnoreturn.h</code> | Adding support for non-returning functions                                               |
| <code>string.h</code>      | Manipulating several kinds of strings                                                    |
| <code>tgmath.h</code>      | Type-generic mathematical functions                                                      |
| <code>threads.h</code>     | Adding support for multiple threads of execution<br>This functionality is not supported. |
| <code>time.h</code>        | Converting between various time and date formats                                         |
| <code>uchar.h</code>       | Unicode functionality                                                                    |
| <code>wchar.h</code>       | Support for wide characters                                                              |
| <code>wctype.h</code>      | Classifying wide characters                                                              |

Table 34: 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 Standard 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 C++:

| Header file                     | Usage                                                                                  |
|---------------------------------|----------------------------------------------------------------------------------------|
| <code>algorithm</code>          | Defines several common operations on containers and other sequences                    |
| <code>array</code>              | Adding support for the array sequencer container                                       |
| <code>atomic</code>             | Adding support for atomic operations<br>This functionality is not supported.           |
| <code>bitset</code>             | Defining a container with fixed-sized sequences of bits                                |
| <code>chrono</code>             | Adding support for time utilities                                                      |
| <code>codecvt</code>            | Adding support for conversions between encodings                                       |
| <code>complex</code>            | Defining a class that supports complex arithmetic                                      |
| <code>condition_variable</code> | Adding support for thread condition variables.<br>This functionality is not supported. |
| <code>deque</code>              | A deque sequence container                                                             |

Table 35: C++ header files

| Header file                   | Usage                                                                                                           |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>exception</code>        | Defining several functions that control exception handling                                                      |
| <code>forward_list</code>     | Adding support for the forward list sequence container                                                          |
| <code>fstream</code>          | Defining several I/O stream classes that manipulate external files                                              |
| <code>functional</code>       | Defines several function objects                                                                                |
| <code>future</code>           | Adding support for passing function information between threads<br>This functionality is not supported.         |
| <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>initializer_list</code> | Adding support for the <code>initializer_list</code> class                                                      |
| <code>iomanip</code>          | Declaring several I/O stream manipulators that take an argument                                                 |
| <code>ios</code>              | Defining the class that serves as the base for many I/O streams classes                                         |
| <code>iosfwd</code>           | Declaring several I/O stream classes before they are necessarily defined                                        |
| <code>iostream</code>         | Declaring the I/O stream objects that manipulate the standard streams                                           |
| <code>istream</code>          | Defining the class that performs extractions                                                                    |
| <code>iterator</code>         | Defines common iterators, and operations on iterators                                                           |
| <code>limits</code>           | Defining numerical values                                                                                       |
| <code>list</code>             | A doubly-linked list sequence container                                                                         |
| <code>locale</code>           | Adapting to different cultural conventions                                                                      |
| <code>map</code>              | A map associative container                                                                                     |
| <code>memory</code>           | Defines facilities for managing memory                                                                          |
| <code>mutex</code>            | Adding support for the data race protection object <code>mutex</code> .<br>This functionality is not supported. |
| <code>new</code>              | Declaring several functions that allocate and free storage                                                      |
| <code>numeric</code>          | Performs generalized numeric operations on sequences                                                            |
| <code>ostream</code>          | Defining the class that performs insertions                                                                     |
| <code>queue</code>            | A queue sequence container                                                                                      |
| <code>random</code>           | Adding support for random numbers                                                                               |
| <code>ratio</code>            | Adding support for compile-time rational arithmetic                                                             |
| <code>regex</code>            | Adding support for regular expressions                                                                          |

Table 35: C++ header files (Continued)

| Header file                   | Usage                                                                                                                  |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------|
| <code>scoped_allocator</code> | Adding support for the memory resource <code>scoped_allocator_adaptor</code>                                           |
| <code>set</code>              | A set associative container                                                                                            |
| <code>shared_mutex</code>     | Adding support for the data race protection object <code>shared_mutex</code> .<br>This functionality is not supported. |
| <code>slist</code>            | A singly-linked list sequence container                                                                                |
| <code>sstream</code>          | Defining several I/O stream classes that manipulate string containers                                                  |
| <code>stack</code>            | A stack sequence container                                                                                             |
| <code>stdexcept</code>        | Defining several classes useful for reporting exceptions                                                               |
| <code>streambuf</code>        | Defining classes that buffer I/O stream operations                                                                     |
| <code>string</code>           | Defining a class that implements a string container                                                                    |
| <code>stringstream</code>     | Defining several I/O stream classes that manipulate in-memory character sequences                                      |
| <code>system_error</code>     | Adding support for global error reporting                                                                              |
| <code>thread</code>           | Adding support for multiple threads of execution.<br>This functionality is not supported.                              |
| <code>tuple</code>            | Adding support for the <code>tuple</code> class                                                                        |
| <code>typeinfo</code>         | Defining type information support                                                                                      |
| <code>typeidindex</code>      | Adding support for type indexes                                                                                        |
| <code>typetraits</code>       | Adding support for traits on types                                                                                     |
| <code>unordered_map</code>    | Adding support for the unordered map associative container                                                             |
| <code>unordered_set</code>    | Adding support for the unordered set associative container                                                             |
| <code>utility</code>          | Defines several utility components                                                                                     |
| <code>valarray</code>         | Defining varying length array container                                                                                |
| <code>vector</code>           | A vector sequence container                                                                                            |

Table 35: C++ header files (Continued)

## Using Standard C libraries in C++

The C++ library works in conjunction with some of the header files from the Standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`. The former puts all declared symbols in the global and `std` namespace, whereas the latter puts them in the global namespace only.

This table shows the new header files:

| <b>Header file</b>       | <b>Usage</b>                                                                              |
|--------------------------|-------------------------------------------------------------------------------------------|
| <code>cassert</code>     | Enforcing assertions when functions execute                                               |
| <code>ccomplex</code>    | Computing common complex mathematical functions                                           |
| <code>cctype</code>      | Classifying characters                                                                    |
| <code>cerrno</code>      | Testing error codes reported by library functions                                         |
| <code>cfenv</code>       | Floating-point exception flags                                                            |
| <code>cfloat</code>      | Testing floating-point type properties                                                    |
| <code>cinttypes</code>   | Defining formatters for all types defined in <code>stdint.h</code>                        |
| <code>ciso646</code>     | Alternative spellings                                                                     |
| <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>cstdalign</code>   | Handling alignment on data objects                                                        |
| <code>cstdarg</code>     | Accessing a varying number of arguments                                                   |
| <code>cstdatomic</code>  | Adding support for atomic operations                                                      |
| <code>cstdbool</code>    | Adds support for the <code>bool</code> data type in C.                                    |
| <code>cstddef</code>     | Defining several useful types and macros                                                  |
| <code>stdint</code>      | Providing integer characteristics                                                         |
| <code>stdio</code>       | Performing input and output                                                               |
| <code>stdlib</code>      | Performing a variety of operations                                                        |
| <code>stdnoreturn</code> | Adding support for non-returning functions                                                |
| <code>cstring</code>     | Manipulating several kinds of strings                                                     |
| <code>ctgmath</code>     | Type-generic mathematical functions                                                       |
| <code>threads</code>     | Adding support for multiple threads of execution.<br>This functionality is not supported. |
| <code>ctime</code>       | Converting between various time and date formats                                          |
| <code>cuchar</code>      | Unicode functionality                                                                     |
| <code>cwchar</code>      | Support for wide characters                                                               |
| <code>cwctype</code>     | Classifying wide characters                                                               |

*Table 36: New Standard C header files—DLIB*



## LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example, `memcpy`, `memset`, and `strcat`.

## NOT SUPPORTED C/C++ FUNCTIONALITY

The following files have contents that are not supported by the IAR C/C++ Compiler:

- `stdatomic.h`, `atomic`
- `threads.h`, `condition_variable`, `future`, `mutex`, `shared_mutex`, `thread`, `threads`
- `exception`, `stdexcept`, `typeinfo`

Some library functions will have the same address. This occurs, most notably, when the library function parameters differ in type but not in size, as for example, `cos(double)` and `cosl(long double)`.

The IAR C/C++ compiler does not support threads as described in the C11 and C++14 standards. However, using `DLib_Threads.h` and an RTOS, you can build an application with thread support. For more information, see *Managing a multithreaded environment*, page 157.

## ATOMIC OPERATIONS

When you compile for cores with instruction set support for atomic accesses, the standard C and C++ atomic operations are available. If atomic operations are not available, the macro `__STDC_NO_ATOMICS__` is defined to 1. This is true both in C and C++.

Atomic operations that cannot be handled natively by the hardware are passed on to library functions. The IAR C/C++ Compiler for RL78 does not include implementations for these functions. A template implementation can be found in the file `src\lib\atomic\libatomic.c`.

## ADDED C FUNCTIONALITY

The DLIB runtime environment includes some added C functionality:

- C bounds-checking interface
- `DLib_Threads.h`
- `fenv.h`
- `LowLevelIOInterface.h`
- `stdio.h`

- `stdlib.h`
- `string.h`
- `time.h` (`time32.h`, `time64.h`)

### C bounds-checking interface

The C library supports Annex K (*Bounds-checking interfaces*) of the C standard. It adds symbols, types, and functions in the header files `errno.h`, `stddef.h`, `stdint.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h` (`time32.h`, `time64.h`), and `wchar.h`.

To enable the interface, define the preprocessor extension `__STDC_WANT_LIB_EXT1__` to 1 prior to including any system header file. See `__STDC_WANT_LIB_EXT1__`, page 392.

As an added benefit, the compiler will issue warning messages for the use of unsafe functions for which the interface has a more safe version. For example, using `strcpy` instead of the more safe `strcpy_s` will make the compiler issue a warning message.

### DLib\_Threads.h

The `DLib_Threads.h` header file contains support for locks and thread-local storage (TLS) variables. This is useful for implementing thread support. For more information, see the header file.

### fenv.h

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapenable` and `fegettrapdisable`.

### LowLevelIOInterface.h

The header file `LowLevelIOInterface.h` contains declarations for the low-level I/O functions used by DLIB. See *The DLIB low-level I/O interface*, page 145.

### stdio.h

These functions provide additional I/O functionality:

|                     |                                                                                     |
|---------------------|-------------------------------------------------------------------------------------|
| <code>fdopen</code> | Opens a file based on a low-level file descriptor.                                  |
| <code>fileno</code> | Gets the low-level file descriptor from the file descriptor ( <code>FILE*</code> ). |
| <code>__gets</code> | Corresponds to <code>fgets</code> on <code>stdin</code> .                           |
| <code>getw</code>   | Gets a <code>wchar_t</code> character from <code>stdin</code> .                     |

|                            |                                                                |
|----------------------------|----------------------------------------------------------------|
| <code>putw</code>          | Puts a <code>wchar_t</code> character to <code>stdout</code> . |
| <code>__ungetchar</code>   | Corresponds to <code>ungetc</code> on <code>stdout</code> .    |
| <code>__write_array</code> | Corresponds to <code>fwrite</code> on <code>stdout</code> .    |

### **string.h**

These are the additional functions defined in `string.h`:

|                          |                                                |
|--------------------------|------------------------------------------------|
| <code>strdup</code>      | Duplicates a string on the heap.               |
| <code>strcasecmp</code>  | Compares strings case-insensitive.             |
| <code>strncasecmp</code> | Compares strings case-insensitive and bounded. |
| <code>strlen</code>      | Bounded string length.                         |

### **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.

The interfaces are defined in three header files:

- `time32.h` defines `__time32_t`, `time_t`, `__time32`, `time`, and associated functions.
- `time64.h` defines `__time64_t`, `time_t`, `__time64`, `time`, and associated functions.
- `time.h` includes `time32.h` or `time64.h` depending on the definition of `_DLIB_TIME_USES_64`.

If `_DLIB_TIME_USES_64` is:

- defined to 1, it will include `time64.h`.
- defined to 0, it will include `time32.h`.
- undefined, it will include `time32.h`.

In both interfaces, `time_t` starts at the year 1970.

An application can use either interface, and even mix them by explicitly using the 32 or 64-bit variants.

See also `__time32`, `__time64`, page 153.

`clock_t` is represented by a 32-bit integer type.

By default, the time library does not support the timezone and daylight saving time functionality. To enable that functionality, use the linker option `--timezone_lib`. See `--timezone_lib`, page 316.

There are two functions that can be used for loading or force-loading the timezone and daylight saving time information from `__getzone`:

- `int _ReloadDstRules (void)`
- `int _ForceReloadDstRules (void)`

Both these functions return 0 for DST rules found and -1 for DST rules not found.

## NON-STANDARD IMPLEMENTATIONS

These functions do not work as specified by the C standard:

- `fopen_s` and `freopen`  
These functions will not propagate the `u` exclusivity attribute to the low-level interface.
- `toupper` and `tolower`  
These functions will only handle A, . . . , Z and a, . . . , z.
- `iswalnum`, . . . , `iswxdigit`  
These functions will only handle arguments in the range 0 to 127.
- The collate functions `strcoll` and `strxfrm` will not work as intended. The same applies to the C++ equivalent functionality.

## SYMBOLS USED INTERNALLY BY THE LIBRARY

The system header files use intrinsic functions, symbols, `pragma` directives etc. Some are defined in the library and some in the compiler. These reserved symbols start with `__` (double underscores) and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

The symbols used internally by the library are not listed in this guide.

# The linker configuration file

- Overview
- Declaring the build type
- Defining memories and regions
- Regions
- Section handling
- Section selection
- Using symbols, expressions, and numbers
- Structural configuration

Before you read this chapter you must be familiar with the concept of *sections*, see *Modules and sections*, page 88.

---

## Overview

To link and locate an application in memory according to your requirements, ILINK needs information about how to handle sections and how to place them into the available memory regions. In other words, ILINK needs a *configuration*, passed to it by means of the *linker configuration file*.

This file consists of a sequence of directives and typically, provides facilities for:

- Declaring the build type  
informing the linker of whether the build is for a traditional ROM system or for a RAM system, helping the linker check that only suitable sections are placed in the different memory regions.
- Defining available addressable memories  
giving the linker information about the maximum size of possible addresses and defining the available physical memory, as well as dealing with memories that can be addressed in different ways.

- Defining the regions of the available memories that are populated with ROM or RAM  
giving the start and end address for each region.
- Section groups  
dealing with how to group sections into blocks and overlays depending on the section requirements.
- Defining how to handle initialization of the application  
giving information about which sections that are to be initialized, and how that initialization should be made.
- Memory allocation  
defining where—in what memory region—each set of sections should be placed.
- Using symbols, expressions, and numbers  
expressing addresses and sizes, etc, in the other configuration directives. The symbols can also be used in the application itself.
- Structural configuration  
meaning that you can include or exclude directives depending on a condition, and to split the configuration file into several different files.
- Special characters in names  
When specifying the name of a symbol or section that uses non-identifier characters, you can enclose the name in back quotes. Example: ``My Name``.

Comments can be written either as C comments (`/* . . . */`) or as C++ comments (`// . . .`).

---

## Declaring the build type

Declaring the build type in the linker configuration files specifies to the linker whether the build is for a traditional ROM system (with, among other things, variable initialization at program start) or for a RAM system to be used for debugging (where other styles of initialization can be used).

## build for directive

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                        |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>build for { ram   rom };</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                        |
| Parameters  | <code>ram</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | The build is assumed to be a debugging or experimental setup, where some or all variable initialization can be performed at load time. |
|             | <code>rom</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | The build is assumed to be a traditional ROM build, where all variable initialization is performed at program start.                   |
| Description | <p>If you declare a build type of <code>rom</code>—and especially if you also declare which memory regions are ROM or RAM—the linker can perform better checking that only suitable sections are placed in the different memory regions. If you do not explicitly specify an <code>initialize</code> directive (see <i>initialize directive</i>, page 418), the linker will behave as if you had specified <code>initialize by copy { rw };</code>.</p> <p>If you declare a build type of <code>ram</code>, the linker does not check which section types are placed in which memory region.</p> <p>If you do not include the <code>build for</code> directive in the linker configuration file, the linker only performs limited checking. This is useful primarily for backward compatibility purposes.</p> |                                                                                                                                        |
| See also    | <i>define region directive</i> , page 408.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                        |

---

## Defining memories and regions

ILINK needs information about the available memory spaces, or more specifically it needs information about:

- The maximum size of possible addressable memories
 

The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. See *define memory directive*, page 408.
- Available physical memory
 

The `define region` directive defines a region in the available memories in which specific sections of application code and sections of application data can be placed. You can also use this directive to declare whether a region contains RAM or ROM memory. This is primarily useful when building for a traditional ROM system. See *define region directive*, page 408.

A region consists of one or several memory ranges. A range is a continuous sequence of bytes in a memory and several ranges can be expressed by using region expressions. See *Region expression*, page 412.

This section gives detailed information about each linker directive specific to defining memories and regions.

## define memory directive

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                  |                                                                                             |                     |                                             |                      |                                                                         |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------------------------------------------------------------------------------------------|---------------------|---------------------------------------------|----------------------|-------------------------------------------------------------------------|
| Syntax               | <pre>define memory [ name ] with size = size_expr [ ,unit-size ];</pre> <p>where <i>unit-size</i> is one of:</p> <pre>unitbitsize = bitsize_expr unitbytesize = bytesize_expr</pre> <p>and where <i>expr</i> is an expression, see <i>expressions</i>, page 433.</p>                                                                                                                                                                                                                                                                                                                                                                                   |                  |                                                                                             |                     |                                             |                      |                                                                         |
| Parameters           | <table> <tr> <td style="vertical-align: top;"><i>size_expr</i></td> <td>Specifies how many <i>units</i> the memory space contains—always counted from address zero.</td> </tr> <tr> <td style="vertical-align: top;"><i>bitsize_expr</i></td> <td>Specifies how many bits each unit contains.</td> </tr> <tr> <td style="vertical-align: top;"><i>bytesize_expr</i></td> <td>Specifies how many bytes each unit contains. Each byte contains 8 bits.</td> </tr> </table>                                                                                                                                                                               | <i>size_expr</i> | Specifies how many <i>units</i> the memory space contains—always counted from address zero. | <i>bitsize_expr</i> | Specifies how many bits each unit contains. | <i>bytesize_expr</i> | Specifies how many bytes each unit contains. Each byte contains 8 bits. |
| <i>size_expr</i>     | Specifies how many <i>units</i> the memory space contains—always counted from address zero.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                  |                                                                                             |                     |                                             |                      |                                                                         |
| <i>bitsize_expr</i>  | Specifies how many bits each unit contains.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                  |                                                                                             |                     |                                             |                      |                                                                         |
| <i>bytesize_expr</i> | Specifies how many bytes each unit contains. Each byte contains 8 bits.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                  |                                                                                             |                     |                                             |                      |                                                                         |
| Description          | <p>The <code>define memory</code> directive defines a <i>memory space</i> with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. This sets the limits for the possible addresses to be used in the linker configuration file. For many microcontrollers, one memory space is sufficient. However, some microcontrollers require two or more. For example, a Harvard architecture usually requires two different memory spaces, one for code and one for data. If only one memory is defined, the memory name is optional. If no <i>unit-size</i> is given, the unit contains 8 bits.</p> |                  |                                                                                             |                     |                                             |                      |                                                                         |
| Example              | <pre>/* Declare the memory space Mem of four Gigabytes */ define memory Mem with size = 4G;</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                  |                                                                                             |                     |                                             |                      |                                                                         |

## define region directive

|        |                                                                                                                                                |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax | <pre>define [ ram   rom ] name = region-expr;</pre> <p>where <i>region-expr</i> is a region expression, see also <i>Regions</i>, page 411.</p> |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------|



|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                 |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| Parameters  | <code>ram</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | The region contains RAM memory. |
|             | <code>rom</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | The region contains ROM memory. |
|             | <code>name</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | The name of the region.         |
| Description | <p>The <code>define region</code> directive defines a region in which specific sections of code and sections of data can be placed. A region consists of one or several memory ranges, where each memory range consists of a continuous sequence of bytes in a specific memory. Several ranges can be combined by using region expressions—these ranges do not need to be consecutive or even in the same memory.</p> <p>If you declare regions as being ROM or RAM, the linker can check that only suitable sections are placed in the regions if you are building a traditional ROM-based system (see <i>build for directive</i>, page 407).</p> |                                 |

|         |                                                                                                                                             |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------|
| Example | <pre>/* Define the 0x10000-byte code region ROM located at address    0x10000 */ define rom region ROM = [from 0x10000 size 0x10000];</pre> |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------|

## logical directive

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                          |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| Syntax      | <pre>logical <i>range-list</i> = physical <i>range-list</i> where <i>range-list</i> is one of [ <i>region-expr</i>, ... ]<i>region-expr</i> [ <i>region-expr</i>, ... ]from <i>address-expr</i></pre>                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                          |
| Parameters  | <code>region-expr</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | A region expression, see also <i>Regions</i> , page 411. |
|             | <code>address-expr</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | An address expression                                    |
| Description | <p>The <code>logical</code> directive maps logical addresses to physical addresses. The physical address is typically used when loading or burning content into memory, while the logical address is the one seen by your application. The physical address is the same as the logical address, if no <code>logical</code> directives are used, or if the address is in a range specified in a <code>logical</code> directive.</p> <p>When generating ELF output, the mapping affects the physical address in program headers. When generating output in the Intel hex or Motorola S-records formats, the physical address is used.</p> |                                                          |

Each address in the logical range list, in the order specified, is mapped to the corresponding address in the physical range list, in the order specified.

Unless one or both of the range lists end with the `from` form, the total size of the logical ranges and the physical ranges must be the same. If one side ends with the `from` form and not the other, the side that ends with the `from` form will include a final range of a size that makes the total sizes match, if possible. If both sides end with a `from` form, the ranges will extend to the highest possible address that makes the total sizes match.

Setting up a mapping from logical to physical addresses can affect how sections and other content are placed. No content will be placed to overlap more than one individual logical or physical range. Also, if there is a mapping from a different logical range to the corresponding physical range, any logical range for which no mapping to physical ranges has been specified—by not being mentioned in a `logical` directive—is excluded from placement.

All `logical` directives are applied together. Using one or using several directives to specify the same mapping makes no difference to the result.

#### Example

```
// Logical range 0x8000-0x8FFF maps to physical 0x10000-0x10FFF.
// No content can be placed in the logical range 0x10000-0x10FFF.
logical [from 0x8000 size 4K] = physical [from 0x10000 size 4K];

// Another way to specify the same mapping
logical [from 0x8000 size 4K] = physical from 0x10000;

// Logical range 0x8000-0x8FFF maps to physical 0x10000-0x10FFF.
// Logical range 0x10000-0x10FFF maps to physical 0x8000-0x8FFF.
// No logical range is excluded from placement because of
// this mapping.
logical [from 0x8000 size 4K] = physical [from 0x10000 size 4K];
logical [from 0x10000 size 4K] = physical [from 0x8000 size 4K];

// Logical range 0x1000-0x13FF maps to physical 0x8000-0x83FF.
// Logical range 0x1400-0x17FF maps to physical 0x9000-0x93FF.
// Logical range 0x1800-0x1BFF maps to physical 0xA000-0xA3FF.
// Logical range 0x1C00-0x1FFF maps to physical 0xB000-0xB3FF.
// No content can be placed in the logical ranges 0x8000-0x83FF,
// 0x9000-0x93FF, 0xA000-0xA3FF, or 0xB000-0xB3FF.
logical [from 0x1000 size 4K] =
 physical [from 0x8000 size 1K repeat 4 displacement 4K];
```

```
// Another way to specify the same mapping.
logical [from 0x1000 to 0x13FF] = physical [from 0x8000 to
0x83FF];
logical [from 0x1400 to 0x17FF] = physical [from 0x9000 to
0x93FF];
logical [from 0x1800 to 0x1BFF] = physical [from 0xA000 to
0xA3FF];
logical [from 0x1C00 to 0x1FFF] = physical [from 0xB000 to
0xB3FF];
```

## Regions

A *region* is a set of non-overlapping memory ranges. A *region expression* is built up out of *region literals* and set operations (union, intersection, and difference) on regions.

### Region literal

#### Syntax

```
[memory-name:] [from expr { to expr | size expr }
```

```
 [repeat expr [displacement expr]]]
```

where *expr* is an expression, see *expressions*, page 433.

#### Parameters

|                          |                                                                                                                                                 |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>memory-name</i>       | The name of the memory space in which the region literal will be located. If there is only one memory, the name is optional.                    |
| from <i>expr</i>         | <i>expr</i> is the start address of the memory range (inclusive).                                                                               |
| to <i>expr</i>           | <i>expr</i> is the end address of the memory range (inclusive).                                                                                 |
| size <i>expr</i>         | <i>expr</i> is the size of the memory range.                                                                                                    |
| repeat <i>expr</i>       | <i>expr</i> defines several ranges in the same memory for the region literal.                                                                   |
| displacement <i>expr</i> | <i>expr</i> is the displacement from the previous range start in the repeat sequence. Default displacement is the same value as the range size. |

#### Description

A region literal consists of one memory range. When you define a range, the memory it resides in, a start address, and a size must be specified. The range size can be stated explicitly by specifying a size, or implicitly by specifying the final address of the range. The final address is included in the range and a zero-sized range will only contain an

address. A range can span over the address zero and such a range can even be expressed by unsigned values, because it is known where the memory wraps.

The `repeat` parameter will create a region literal that contains several ranges, one for each repeat. This is useful for *banked* or *far* regions.

### Example

```
/* The 5-byte size range spans over the address zero */
Mem:[from -2 to 2]

/* The 512-byte size range spans over zero, in a 64-Kbyte memory */
Mem:[from 0xFF00 to 0xFF]

/* Defining several ranges in the same memory, a repeating
 literal */
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]

/* Resulting in a region containing:
 Mem:[from 0 size 0x100]
 Mem:[from 0x1000 size 0x100]
 Mem:[from 0x2000 size 0x100]
*/
```

See also

*define region directive*, page 408, and *Region expression*, page 412.

## Region expression

### Syntax

```
region-operand
| region-expr | region-operand
| region-expr - region-operand
| region-expr & region-operand
```

where *region-operand* is one of:

```
(region-expr)
region-name
region-literal
empty-region
```

where *region-name* is a region, see *define region directive*, page 408

where *region-literal* is a region literal, see *Region literal*, page 411

and where *empty-region* is an empty region, see *Empty region*, page 413.

### Description

Normally, a region consists of one memory range, which means a *region literal* is sufficient to express it. When a region contains several ranges, possibly in different

memories, it is instead necessary to use a *region expression* to express it. Region expressions are actually set expressions on sets of memory ranges.

To create region expressions, three operators are available: union (`|`), intersection (`&`), and difference (`-`). These operators work as in *set theory*. For example, if you have the sets A and B, then the result of the operators would be:

- A | B: all elements in either set A or set B
- A & B: all elements in both set A and B
- A - B: all elements in set A but not in B.

### Example

```
/* Resulting in a range starting at 1000 and ending at 2FFF, in
 memory Mem */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1500 and ending at 1FFF, in
 memory Mem */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1000 and ending at 14FF, in
 memory Mem */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]

/* Resulting in two ranges. The first starting at 1000 and ending
 at 1FFF, the second starting at 2501 and ending at 2FFF.
 Both located in memory Mem */
Mem:[from 0x1000 to 0x2FFF] - Mem:[from 0x2000 to 0x24FF]
```

## Empty region

Syntax

```
[]
```

Description

The empty region does not contain any memory ranges. If the empty region is used in a placement directive that actually is used for placing one or more sections, ILINK will issue an error.

**Example**

```

define region Code = Mem:[from 0 size 0x10000];
if (Banked) {
 define region Bank = Mem:[from 0x8000 size 0x1000];
} else {
 define region Bank = [];
}
define region NonBanked = Code - Bank;

/* Depending on the Banked symbol, the NonBanked region is either
one range with 0x10000 bytes, or two ranges with 0x8000 and
0x7000 bytes, respectively. */

```

**See also**

*Region expression*, page 412.

---

## Section handling

Section handling describes how ILINK should handle the sections of the execution image, which means:

- **Placing sections in regions**

The `place at` and `place in` directives place sets of sections with similar attributes into previously defined regions. See *place at directive*, page 422 and *place in directive*, page 423.
- **Making sets of sections with special requirements**

The `block` directive makes it possible to create empty sections with specific or expanding sizes, specific alignments, sequentially sorted sections of different types, etc.

The `overlay` directive makes it possible to create an area of memory that can contain several overlay images. See *define block directive*, page 415, and *define overlay directive*, page 417.
- **Initializing the application**

The directives `initialize` and `do not initialize` control how the application should be started. With these directives, the application can initialize global symbols at startup, and copy pieces of code. The initializers can be stored in several ways, for example, they can be compressed. See *initialize directive*, page 418 and *do not initialize directive*, page 421.
- **Keeping removed sections**

The `keep` directive retains sections even though they are not referred to by the rest of the application, which means it is equivalent to the *root* concept in the assembler and compiler. See *keep directive*, page 421.

- Specifying the contents of linker-generated sections  
The `define section` directive can be used for creating specific sections with content and calculations that are only available at link time.
- Additional more specialized directives:  
`use init format directive`, `use init table directive`

This section gives detailed information about each linker directive specific to section handling.

## define block directive

### Syntax

```
define block name
 [with param, param...]
{
 extended-selectors
}
[except
{
 section-selectors
}];
```

where *param* can be one of:

```
size = expr
minimum size = expr
maximum size = expr
expanding size
alignment = expr
fixed order
alphabetical order
```

and where the rest of the directive selects sections to include in the block, see *Section selection*, page 426.

### Parameters

|                     |                                                                                                                                           |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <i>name</i>         | The name of the block to be defined.                                                                                                      |
| <i>size</i>         | Customizes the size of the block. By default, the size of a block is the sum of its parts dependent of its contents.                      |
| <i>minimum size</i> | Specifies a lower limit for the size of the block. The block is at least this large, even if its contents would otherwise not require it. |
| <i>maximum size</i> | Specifies an upper limit for the size of the block. An error is generated if the sections in the block do not fit.                        |

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| expanding size     | The block will expand to use all available space in the memory range where it is placed.                                                                                                                                                                                                                                                                                                                                                                                                   |
| alignment          | Specifies a minimum alignment for the block. If any section in the block has a higher alignment than the minimum alignment, the block will have that alignment.                                                                                                                                                                                                                                                                                                                            |
| fixed order        | Places sections in the specified order. Each <i>extended-selector</i> is added in a separate nested block, and these blocks are kept in the specified order.                                                                                                                                                                                                                                                                                                                               |
| alphabetical order | Places sections in alphabetical order by section name. Only <i>section-selector</i> patterns are allowed in <code>alphabetical order</code> blocks, for example, no nested blocks. All sections in a particular alphabetical order block must use the same kind of initialization (read-only, zero-init, copy-init, or no-init, and otherwise equivalent). You cannot use <code>__section_begin</code> , etc on individual sections contained in an <code>alphabetical order</code> block. |

**Description**

The `block` directive defines a contiguous area of memory that contains a possibly empty set of sections or other blocks. Blocks with no content are useful for allocating space for stacks or heaps. Blocks with content are usually used to group together sections that must to be consecutive.

You can access the start, end, and size of a block from an application by using the `__section_begin`, `__section_end`, or `__section_size` operators. If there is no block with the specified name, but there are sections with that name, a block will be created by the linker, containing all such sections.

Blocks with expanding size are most often used for heaps or stacks.

**Note:** You cannot place a block with expanding size inside another block with expanding size, inside a block with a maximum size, or inside an overlay.

**Example**

```
/* Create a block with a minimum size for the heap that will use
all remaining space in its memory range */
define block HEAP with minimum size = 4K, expanding size,
alignment = { };
```

**See also**

*Interaction between the tools and your application*, page 207. For an accessing example, see *define overlay directive*, page 417.



## define overlay directive

```
Syntax define overlay name [with param, param...]
 {
 extended-selectors;
 }
 [except
 {
 section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 426.

### Parameters

|                           |                                                                                                                                                                       |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>name</code>         | The name of the overlay.                                                                                                                                              |
| <code>size</code>         | Customizes the size of the overlay. By default, the size of a overlay is the sum of its parts dependent of its contents.                                              |
| <code>maximum size</code> | Specifies an upper limit for the size of the overlay. An error is generated if the sections in the overlay do not fit.                                                |
| <code>alignment</code>    | Specifies a minimum alignment for the overlay. If any section in the overlay has a higher alignment than the minimum alignment, the overlay will have that alignment. |
| <code>fixed order</code>  | Places sections in fixed order—if not specified, the order of the sections will be arbitrary.                                                                         |

### Description

The `overlay` directive defines a named set of sections. In contrast to the `block` directive, the `overlay` directive can define the same name several times. Each definition will then be grouped in memory at the same place as all other definitions of the same name. This creates an *overlaid* memory area, which can be useful for an application that has several independent sub-applications.

Place each sub-application image in ROM and reserve a RAM overlay area that can hold all sub-applications. To execute a sub-application, first copy it from ROM to the RAM overlay.

**Note:** ILINK does not help you with managing interdependent overlay definitions, apart from generating a diagnostic message for any reference from one overlay to another overlay.

The size of an overlay will be the same size as the largest definition of that overlay name and the alignment requirements will be the same as for the definition with the highest alignment requirements.

**Note:** Sections that were overlaid must be split into a RAM and a ROM part and you must take care of all the copying needed.



Code in overlaid memory areas cannot be debugged; the C-SPY Debugger cannot determine which code is currently loaded.

See also

*Manual initialization*, page 111.

## initialize directive

### Syntax

```
initialize { by copy | manually }
 [with param, param...]
{
 section-selectors
}
[except
 {
 section-selectors
 }];
```

where *param* can be one of:

```
packing = algorithm
simple ranges
complex ranges
no exclusions
```

For information about section selectors and except clauses, see *Section selection*, page 426.

### Parameters

|                       |                                                                                                                                                 |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>by copy</code>  | Splits the section into sections for initializers and initialized data, and handles the initialization at application startup automatically.    |
| <code>manually</code> | Splits the section into sections for initializers and initialized data. The initialization at application startup is not handled automatically. |

*algorithm*

Specifies how to handle the initializers. Choose between:

*none* - Disables compression of the selected section contents. This is the default method for *initialize manually*.

*zeros* - Compresses consecutive bytes with the value zero.

*packbits* - Compresses with the PackBits algorithm. This method generates good results for data with many identical consecutive bytes.

*lz77* - Compresses with the Lempel-Ziv-77 algorithm. This method handles a larger variety of inputs well, but has a slightly larger decompressor.

*auto* - ILINK estimates the resulting size using each packing method (except for *auto*), and then chooses the packing method that produces the smallest estimated size. Note that the size of the decompressor is also included. This is the default method for *initialize by copy*.

*smallest* - This is a synonym for *auto*.

## Description

The *initialize* directive splits each selected section into one section that holds initializer data and another section that holds the space for the initialized data. The section that holds the space for the initialized data retains the original section name, and the section that holds initializer data gets the name suffix *\_init*. You can choose whether the initialization at startup should be handled automatically (*initialize by copy*) or whether you should handle it yourself (*initialize manually*).

When you use the packing method *auto* (default for *initialize by copy*), ILINK will automatically choose an appropriate packing algorithm for the initializers. To override this, specify a different *packing method*. The *--log initialization* option shows how ILINK decided which packing algorithm to use.

When initializers are compressed, a decompressor is automatically added to the image.

Each decompressor has two variants: one that can only handle a single source and destination range at a time, and one that can handle more complex cases. By default, the linker chooses a decompressor variant based on whether the associated section placement directives specify a single or multi-range memory region. In general, this is the desired behavior, but you can use the *with complex ranges* or the *with simple ranges* modifier on an *initialize* directive to specify which decompressor variant to use. You can also use the command line option *--default\_to\_complex\_ranges* to make *initialize* directives by default use complex ranges. The *simple ranges* decompressors are normally hundreds of bytes smaller than the *complex ranges* variants.

When initializers are compressed, the exact size of the compressed initializers is unknown until the exact content of the uncompressed data is known. If this data contains other addresses, and some of these addresses are dependent on the size of the compressed initializers, the linker fails with error Lp017. To avoid this, place compressed initializers last, or in a memory region together with sections whose addresses do not need to be known.

Due to an internal dependence, generation of compressed initializers can also fail (with error LP021) if the address of the initialized area depends on the size of its initializers. To avoid this, place the initializers and the initialized area in different parts of the memory (for example, the initializers are placed in ROM and the initialized area in RAM).

If you specify the parameter `no_exclusions`, an error is emitted if any sections are excluded (because they are needed for the initialization). `no_exclusions` can only be used with `initialize by copy` (automatic initialization), not with `initialize manually`.

Unless `initialize manually` is used, ILINK will arrange for initialization to occur during system startup by including an initialization table. Startup code calls an initialization routine that reads this table and performs the necessary initializations.

Zero-initialized sections are not affected by the `initialize` directive.

The `initialize` directive is normally used for initialized variables, but can be used for copying any sections, for example, copying executable code from slow ROM to fast RAM, or for overlays. For another example, see *define overlay directive*, page 417.

Sections that are needed for initialization are not affected by the `initialize by copy` directive. This includes the `__low_level_init` function and anything it references.

Anything reachable from the program entry label is considered *needed for initialization* unless reached via a section fragment with a label starting with `__iar_init$$done`. The `--log sections` option, in addition to logging the marking of section fragments to be included in the application, also logs the process of determining which sections are needed for initialization.

#### Example

```
/* Copy all read-write sections automatically from ROM to RAM at
 program start */
initialize by copy { rw };
place in RAM { rw };
place in ROM { ro };
```

#### See also

*Initialization at system startup*, page 94, and *do not initialize directive*, page 421.

## do not initialize directive

Syntax

```
do not initialize
{
 section-selectors
}
[except
{
 section-selectors
}];
```

For information about section selectors and except clauses, see *Section selection*, page 426.

Description

Use the `do not initialize` directive to specify the sections that you do not want to be automatically zero-initialized by the system startup code. The directive can only be used on `zeroinit` sections.

Typically, this is useful if you want to handle zero-initialization in some other way for all or some `zeroinit` sections.

This can also be useful if you want to suppress zero-initialization of variables entirely. Normally, this is handled automatically for variables specified as `__no_init` in the source, but if you link with object files produced by older tools from IAR Systems or other tool vendors, you might need to suppress zero-initialization specifically for some sections.

Example

```
/* Do not initialize read-write sections whose name ends with
 _noinit at program start */
do not initialize { rw section .*_noinit };
place in RAM { rw section .*_noinit };
```

See also *Initialization at system startup*, page 94, and *initialize directive*, page 418.

## keep directive

Syntax

```
keep
{
 [{ section-selectors | block name }
 [, {section-selectors | block name }...]]
}
[except
{
 section-selectors
}];
```

For information about selectors and except clauses, see *Section selection*, page 426.

**Description** The `keep` directive can be used for including blocks, overlays, or sections in the executable image that would otherwise be discarded because no references to them exist in the included parts of the application. Note that only sections from included modules are considered for inclusion.

The `keep` directive does not cause any additional *modules* to be included in the application. To cause modules that define the specified symbols to be included, use the **Keep symbols** linker option (or the `--keep` command line option).

**Example**

```
keep { section .keep* } except {section .keep};
```

## place at directive

**Syntax**

```
["name":]
place [noload] at { address [memory:] address |
 start of region_expr [with mirroring to mirror_address] |
 end of region_expr [with mirroring to mirror_address] }

{
 extended-selectors
}
[except
{
 section-selectors
}];
```

For information about extended selectors and except clauses, see *Section selection*, page 426.

### Parameters

|               |                                                                                                                                                                                                                                                                  |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>name</i>   | Optional. If it is specified, it is used in the map file, in some log messages, and is part of the name of any ELF output sections resulting from the directive.                                                                                                 |
| <i>noload</i> | Optional. If it is specified, it prevents the sections in the directive from being loaded to the target system. To use the sections, you must put them into the target system in some other way. <i>noload</i> can only be used when a <i>name</i> is specified. |

|                             |                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>memory: address</i>      | A specific address in a specific memory. The address must be available in the supplied memory defined by the <code>define memory</code> directive. The memory specifier is optional if there is only one memory.                                                                                                                                                                                        |
| start of <i>region_expr</i> | A region expression that results in a single-internal region. The start of the interval is used.                                                                                                                                                                                                                                                                                                        |
| end of <i>region_expr</i>   | A region expression that results in a single-internal region. The end of the interval is used.                                                                                                                                                                                                                                                                                                          |
| <i>mirror_address</i>       | If with <code>mirroring to</code> is specified, the contents of any sections are assumed to be mirrored to this address, therefore debug information and symbols will appear in the mirrored range, but the actual content bytes are placed as if with <code>mirroring to</code> was not specified.<br><br><b>Note:</b> This functionality is intended to support external (target-specific) mirroring. |

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | The <code>place at</code> directive places sections and blocks either at a specific address or, at the beginning or the end of a region. The same address cannot be used for two different <code>place at</code> directives. It is also not possible to use an empty region in a <code>place at</code> directive. If placed in a region, the sections and blocks will be placed before any other sections or blocks placed in the same region with a <code>place in</code> directive.<br><br><b>Note:</b> with <code>mirroring to</code> can be used only together with <code>start of</code> and <code>end of</code> . |
| Example     | <pre>/* Place the RO section .startup at the start of code_region */ "START": place at start of ROM { readonly section .startup };</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| See also    | <i>place in directive</i> , page 423.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

## place in directive

|        |                                                                                                                                                                                             |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax | <pre>[ "name": ] place [ noload ] in <i>region_expr</i>     [ with mirroring to <i>mirror_address</i> ] {     <i>extended-selectors</i> } [ except{     <i>section-selectors</i> } ];</pre> |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

where *region-expr* is a region expression, see also *Regions*, page 411.

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 426.

#### Parameters

|                       |                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>name</i>           | Optional. If it is specified, it is used in the map file, in some log messages, and is part of the name of any ELF output sections resulting from the directive.                                                                                                                                                                                                                            |
| <i>noload</i>         | Optional. If it is specified, it prevents the sections in the directive from being loaded to the target system. To use the sections, you must put them into the target system in some other way. <i>noload</i> can only be used when a <i>name</i> is specified.                                                                                                                            |
| <i>mirror_address</i> | If with <i>mirroring to</i> is specified, the contents of any sections are assumed to be mirrored to this address, therefore debug information and symbols will appear in the mirrored range, but the actual content bytes are placed as if with <i>mirroring to</i> was not specified.<br><br><b>Note:</b> This functionality is intended to support external (target-specific) mirroring. |

#### Description

The *place in* directive places sections and blocks in a specific region. The sections and blocks will be placed in the region in an arbitrary order.

To specify a specific order, use the *block* directive. The region can have several ranges.

**Note:** When with *mirroring to* is specified, the *region-expr* must result in a single range.

#### Example

```
/* Place the read-only sections in the code_region */
"ROM": place in ROM { readonly };
```

#### See also

*place at directive*, page 422.



## use init format directive

Syntax

```
use init format name for
{
 section-selectors
}
[except
{
 section-selectors
}];
```

For information about extended selectors and except clauses, see *Section selection*, page 426.

### Parameters

|             |                                                                              |
|-------------|------------------------------------------------------------------------------|
| <i>name</i> | The name of the format to use.                                               |
|             | Specify <i>near</i> to generate smaller entries in the initialization table. |

### Description

Entries in the initialization table can use different formats, typically depending on the range of representable addresses in the selected format. The linker uses a default format for initialization table entries to control the initialization of sections that do not match any `use init format` directive.

Use this directive for explicitly selecting a format resulting in smaller entries for some initializations, or to select a format that can handle larger addresses than the default.

### Example

```
use init format near for { section *.near};
```

## use init table directive

Syntax

```
use init table name for
{
 section-selectors
}
[except
{
 section-selectors
}];
```

For information about section selectors and except clauses, see *Section selection*, page 426.

## Parameters

*name* The name of the `init` table.

## Description

Normally, all initialization entries are generated into a single initialization table (called `Table`). Use this directive to cause some of the entries to be put into a separate table. You can then use this initialization table at another time, or under different circumstances, than the normal initialization table.

Initialization entries for all variables not mentioned in a `use init table` directive are put into the normal initialization table. By having multiple `use init table` directives you can have multiple initialization tables.

The start, end, and size of the `init` table can be accessed in the application program by using `__section_begin`, `__section_end`, or `__section_size` of `"Region$$name"`, respectively, or via the symbols `Region$$name$$Base`, `Region$$name$$Limit`, and `Region$$name$$Length`.

## Example

```
use init table Core2 for { section *.core2};

/* __section_begin("Region$$Core2") can be used to get the start
 of the Core2 init table. */
```

---

## Section selection

The purpose of *section selection* is to specify—by means of *section selectors* and *except clauses*—the sections that an `ILINK` directive should be applied to. All sections that match one or more of the section selectors will be selected, and none of the sections selectors in the `except` clause, if any. Each section selector can match sections on section attributes, section name, and object or library name.

Some directives provide functionality that requires more detailed selection capabilities, for example, directives that can be applied on both sections and blocks. In this case, the *extended-selectors* are used.

This section gives detailed information about each linker directive specific to section selection.

## section-selectors

### Syntax

```
[section-selector [, section-selector...]]
section-selector is:
 [section-attribute] [section-type]
 [symbol symbol-name] [section section-name]
 [object module-spec]
section-attribute is:
 ro [code | data] | rw [code | data] | zi
section-type is:
 [preinit_array | init_array]
```

### Parameters

*section-attribute* Only sections with the specified attribute will be selected. *section-attribute* can consist of:

- ro|readonly, for ROM sections.
- rw|readwrite, for RAM sections.

In each category, sections can be further divided into those that contain code and those that contain data, resulting in four main categories:

- ro code, for normal code
- ro data, for constants
- rw code, for code copied to RAM
- rw data, for variables

readwrite data also has a subcategory—*zi*|zeroinit—for sections that are zero-initialized at application startup.

*section-type* Only sections with that ELF section type will be selected. *section-type* can be:

- preinit\_array, sections of the ELF section type SHT\_PREINIT\_ARRAY.
- init\_array, sections of the ELF section type SHT\_INIT\_ARRAY.

|                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>symbol</code> <i>symbol-name</i>   | <p>Only sections that define at least one public symbol that matches the symbol name pattern will be selected. <i>symbol-name</i> is the symbol name pattern. Two wildcards are allowed:</p> <p>? matches any single character.<br/>* matches zero or more characters.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>section</code> <i>section-name</i> | <p>Only sections whose names match the <i>section-name</i> will be selected. Two wildcards are allowed:</p> <p>? matches any single character<br/>* matches zero or more characters.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>object</code> <i>module-spec</i>   | <p>Only sections that originate from library modules or object files that matches <i>module-spec</i> will be selected. <i>module-spec</i> can be in one of two forms:</p> <p><i>module</i>, a name in the form <i>objectname(libraryname)</i>. Sections from object modules where both the object name and the library name match their respective patterns are selected. An empty library name pattern selects only sections from object files. If <i>libraryname</i> is <i>:sys</i>, the pattern will match only sections from the system library.</p> <p><i>filename</i>, the name of an object file, or an object in a library.</p> <p>Two wildcards are allowed:</p> <p>? matches any single character<br/>* matches zero or more characters.</p> |

**Description**

A section selector selects all sections that match the section attribute, section type, symbol name, section name, and the name of the module. Up to four of the five conditions can be omitted.

It is also possible to use only { } without any section selectors, which can be useful when defining blocks.

**Note:** A section selector with narrower scope has higher priority than a more generic section selector. If more than one section selector matches for the same purpose, one of them must be more specific. A section selector is more specific than another one if in priority order:

- It specifies a symbol name with no wildcards and the other one does not.

- It specifies a section name or object name with no wildcards and the other one does not
- It specifies a section type and the other one does not
- There could be sections that match the other selector that also match this one, however, the reverse is not true.

| Selector 1         | Selector 2       | More specific |
|--------------------|------------------|---------------|
| ro                 | ro code          | Selector 2    |
| symbol mysym       | section foo      | Selector 1    |
| ro code section f* | ro section f*    | Selector 1    |
| section foo*       | section f*       | Selector 1    |
| section *x         | section f*       | Neither       |
| init_array         | section f*       | Selector 1    |
| section .intvec    | ro section .int* | Selector 1    |
| section .intvec    | object foo.o     | Neither       |

Table 37: Examples of section selector specifications

#### Example

```
{ rw } /* Selects all read-write sections */

{ section .mydata* } /* Selects only .mydata* sections */
/* Selects .mydata* sections available in the object special.o */
{ section .mydata* object special.o }
```

Assuming a section in an object named `foo.o` in a library named `lib.a`, any of these selectors will select that section:

```
object foo.o(lib.a)
object f*(lib*)
object foo.o
object lib.a
```

#### See also

*initialize directive*, page 418, *do not initialize directive*, page 421, and *keep directive*, page 421.

## extended-selectors

### Syntax

```
[extended-selector [, extended-selector...]]
where extended-selector is:
 [first | last | midway]
 { section-selector |
 block name [inline-block-def] |
 overlay name }
where inline-block-def is:
 [block-params] extended-selectors
```

### Parameters

|               |                                                                                                                                                                                                                                                      |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>first</i>  | Places the selected sections, block, or overlay first in the containing placement directive, block, or overlay.                                                                                                                                      |
| <i>last</i>   | Places the selected sections, block or overlay last in the containing placement directive, block, or overlay.                                                                                                                                        |
| <i>midway</i> | Places the selected sections, block, or overlay so that they are no further than half the maximum size of the containing block away from either edge of the block. Note that this parameter can only be used inside a block that has a maximum size. |
| <i>name</i>   | The name of the block or overlay.                                                                                                                                                                                                                    |

### Description

Use *extended-selectors* to select content for inclusion in a placement directive, block, or overlay. In addition to using section selection patterns, you can also explicitly specify blocks or overlays for inclusion.

Using the *first* or *last* keyword, you can specify one pattern, block, or overlay that is to be placed first or last in the containing placement directive, block, or overlay. If you need more precise control of the placement order you can instead use a block with fixed order.

Blocks can be defined separately, using the `define block` directive, or `inline`, as part of an *extended-selector*.

The *midway* parameter is primarily useful together with a static base that can have both negative and positive offsets.

**Example**

```
define block First { ro section .f* }; /* Define a block holding
 any read-only section*/
 matching ".f*" */
define block Table { first block First, ro section .b };
 /* Define a block where
 the block First comes
 before the sections
 matching ".b*". */
```

You can also define the block `First` inline, instead of in a separate `define block` directive:

```
define block Table { first block First { ro section .f* },
 ro section .b* };
```

**See also**

*define block directive*, page 415, *define overlay directive*, page 417, and *place at directive*, page 422.

---

## Using symbols, expressions, and numbers

In the linker configuration file, you can also:

- *Define and export symbols*

The `define symbol` directive defines a symbol with a specified value that can be used in expressions in the configuration file. The symbol can also be exported to be used by the application or the debugger. See *define symbol directive*, page 432, and *export directive*, page 433.

- *Use expressions and numbers*

In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, etc. See *expressions*, page 433.

This section gives detailed information about each linker directive specific to defining symbols, expressions and numbers.

### check that directive

**Syntax**

```
check that expression;
```

**Parameters**

*expression*

A boolean expression.

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>You can use the <code>check that</code> 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.</p> <p>Three extra operators are available for use only in <code>check that</code> expressions:</p> <p><code>maxstack(category)</code>      The stack depth of the deepest call chain for any call graph root function in the category.</p> <p><code>totalstack(category)</code>      The sum of the stack depths of the deepest call chains for each call graph root function in the category.</p> <p><code>size(block)</code>      The size of the block.</p> |
| Example     | <pre>check that maxstack("Program entry")            + totalstack("interrupt")            + 1K            &lt;= size(block CSTACK);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| See also    | <i>Stack usage analysis</i> , page 97.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## define symbol directive

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>define [ exported ] symbol name = expr;</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Parameters  | <p><code>exported</code>      Exports the symbol to be usable by the executable image.</p> <p><code>name</code>      The name of the symbol.</p> <p><code>expr</code>      The symbol value.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description | <p>The <code>define symbol</code> directive defines a symbol with a specified value. The symbol can then be used in expressions in the configuration file. The symbols defined in this way work exactly like the symbols defined with the option <code>--config_def</code> outside of the configuration file.</p> <p>The <code>define exported symbol</code> variant of this directive is a shortcut for using the directive <code>define symbol</code> in combination with the <code>export symbol</code> directive. On the command line this would require both a <code>--config_def</code> option and a <code>--define_symbol</code> option to achieve the same effect.</p> <p><b>Note:</b></p> <ul style="list-style-type: none"> <li>● A symbol cannot be redefined</li> </ul> |





where *symbol* is a defined symbol, see *define symbol directive*, page 432 and *--config\_def*, page 295

and where *func-operator* is one of these function-like operators:

|                                                   |                                                                       |
|---------------------------------------------------|-----------------------------------------------------------------------|
| <code>minimum(<i>expr</i>, <i>expr</i>)</code>    | Returns the smallest of the two parameters.                           |
| <code>maximum(<i>expr</i>, <i>expr</i>)</code>    | Returns the largest of the two parameters.                            |
| <code>isempty(<i>r</i>)</code>                    | Returns True if the region is empty, otherwise False.                 |
| <code>isdefinedsymbol(<i>expr-symbol</i> )</code> | Returns True if the expression symbol is defined, otherwise False.    |
| <code>start(<i>r</i>)</code>                      | Returns the lowest address in the region.                             |
| <code>end(<i>r</i>)</code>                        | Returns the highest address in the region.                            |
| <code>size(<i>r</i>)</code>                       | Returns the size of the complete region.                              |
| <code>isbuildfor( { ram   rom } )</code>          | Returns True if the build type is the one specified, otherwise False. |

where *expr* is an expression, and *r* is a region expression, see *Region expression*, page 412.

## Description

In the linker configuration file, an expression is a 65-bit value with the range  $-2^{64}$  to  $2^{64}$ . The expression syntax closely follows C syntax with some minor exceptions. There are no assignments, casts, pre or post-operations, and no address operations (`*`, `&`, `[]`, `->`, and `.`). Some operations that extract a value from a region expression, etc, use a syntax resembling that of a function call. A boolean expression returns 0 (False) or 1 (True).

## numbers

### Syntax

`nr [nr-suffix]`

where *nr* is either a decimal number or a hexadecimal number (0x... or 0X...).

and where *nr-suffix* is one of:

```
K /* Kilo = (1 << 10) 1024 */
M /* Mega = (1 << 20) 1048576 */
G /* Giga = (1 << 30) 1073741824 */
T /* Tera = (1 << 40) 1099511627776 */
P /* Peta = (1 << 50) 1125899906842624 */
```

|             |                                                                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | A number can be expressed either by normal C means or by suffixing it with a set of useful suffixes, which provides a compact way of specifying numbers. |
| Example     | 1024 is the same as 0x400, which is the same as 1K.                                                                                                      |

---

## Structural configuration

The structural directives provide means for creating structure within the configuration, such as:

- Conditional inclusion  
An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations in the same file. See *if directive*, page 436.
- Dividing the linker configuration file into several different files  
The `include` directive makes it possible to divide the configuration file into several logically distinct files. See *include directive*, page 436.
- Signaling an error for unsupported cases

This section gives detailed information about each linker directive specific to structural configuration.

### error directive

|             |                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>error string</code>                                                                                                                 |
| Parameters  | <code>string</code> The error message.                                                                                                    |
| Description | An <code>error</code> directive can be used for signaling an error if the directive occurs in the active part of a conditional directive. |
| Example     | <code>error "Unsupported configuration"</code>                                                                                            |

## if directive

### Syntax

```
if (expr) {
 directives
[} else if (expr) {
 directives]
[} else {
 directives]
}
```

where *expr* is an expression, see *expressions*, page 433.

### Parameters

*directives*                      Any ILINK directive.

### Description

An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations, for example, both a banked and non-banked memory configuration, in the same file.

The text inside a non-selected part of an `if` directive is not checked for syntax. The only requirements for such text, is that it can be tokenized, and that any open brace (`{`) token has a matching close brace (`}`) token.

### Example

See *Empty region*, page 413.

## include directive

### Syntax

```
include "filename";
```

### Parameters

*filename*                      A path where both `/` and `\` can be used as the directory delimiter.

### Description

The `include` directive makes it possible to divide the configuration file into several logically distinct parts, each in a separate file. For instance, there might be parts that you need to change often and parts that you seldom edit.

Normally, the linker searches for configuration include files in the system configuration directory. You can use the `--config_search` linker option to add more directories to search.

### See also

`--config_search`, page 295

# Section reference

- Summary of sections
- Descriptions of sections and blocks

For more information, see the chapter *Modules and sections*, page 88.

---

## Summary of sections

The compiler places code and data into sections. Based on a configuration specified in the linker configuration file, ILINK places sections in memory.

This table lists the ELF sections and blocks that are used by the IAR build tools:

| Section                     | Description                                                                                                      |
|-----------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>.bss</code>           | Holds zero-initialized <code>__near</code> static and global variables.                                          |
| <code>.bss.noinit</code>    | Holds <code>__no_init __near</code> static and global variables.                                                 |
| <code>.bssf</code>          | Holds zero-initialized <code>__far</code> static and global variables.                                           |
| <code>.bssf.noinit</code>   | Holds <code>__no_init __far</code> static and global variables.                                                  |
| <code>.bss_unit64kp</code>  | This section is used internally by the linker.                                                                   |
| <code>.callt0</code>        | Holds the call table vectors generated by use of the <code>__callt</code> extended keyword.                      |
| <code>.const</code>         | Holds <code>__near</code> constant data.                                                                         |
| <code>.constf</code>        | Holds <code>__far</code> constant data.                                                                          |
| <code>.consth</code>        | Holds <code>__huge</code> constant data.                                                                         |
| <code>CSTACK</code>         | Holds the stack used by C or C++ programs.                                                                       |
| <code>.data</code>          | Holds <code>__near</code> static and global initialized variables.                                               |
| <code>.data_init</code>     | Holds initial values for <code>.data</code> sections when the linker directive <code>initialize</code> is used.  |
| <code>.dataf</code>         | Holds <code>__far</code> static and global initialized variables.                                                |
| <code>.dataf_init</code>    | Holds initial values for <code>.dataf</code> sections when the linker directive <code>initialize</code> is used. |
| <code>.data_unit64kp</code> | This section is used internally by the linker.                                                                   |
| <code>FAR_HEAP</code>       | Holds the heap used for dynamically allocated <code>__far</code> data.                                           |
| <code>.hbss</code>          | Holds zero-initialized <code>__huge</code> static and global variables.                                          |

Table 38: Section summary

| Section                         | Description                                                                                                      |
|---------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>.hbss.noinit</code>       | Holds <code>__no_init __huge</code> static and global variables.                                                 |
| <code>.hdata</code>             | Holds <code>__huge</code> static and global initialized variables.                                               |
| <code>.hdata_init</code>        | Holds initial values for <code>.hdata</code> sections when the linker directive <code>initialize</code> is used. |
| <code>HUGE_HEAP</code>          | Holds the heap used for dynamically allocated <code>__huge</code> data.                                          |
| <code>.iar.dynexit</code>       | Holds the <code>atexit</code> table.                                                                             |
| <code>__iar_tls.\$\$DATA</code> | Holds initial values for TLS variables.                                                                          |
| <code>.init_array</code>        | Holds a table of dynamic initialization functions.                                                               |
| <code>.intvec</code>            | Holds the interrupt vector table                                                                                 |
| <code>NEAR_HEAP</code>          | Holds the heap used for dynamically allocated <code>__near</code> data.                                          |
| <code>.option_byte</code>       | Holds the OCD option bytes for configuring the on-chip debugging interface.                                      |
| <code>.preinit_array</code>     | Holds a table of dynamic initialization functions.                                                               |
| <code>.sbss</code>              | Holds zero-initialized static and global variables.                                                              |
| <code>.sbss.noinit</code>       | Holds <code>__no_init __saddr</code> static and global variables.                                                |
| <code>.sdata</code>             | Holds <code>__saddr</code> static and global initialized variables.                                              |
| <code>.sdata_init</code>        | Holds initial values for <code>.sdata</code> sections when the linker directive <code>initialize</code> is used. |
| <code>.security_id</code>       | Holds the Security ID that allows an authentication check before a debug session is started.                     |
| <code>.switch</code>            | Holds switch tables for <code>__near_func</code> functions.                                                      |
| <code>.switchf</code>           | Holds switch tables for <code>__far_func</code> functions.                                                       |
| <code>.text</code>              | Holds <code>__near_func</code> functions, interrupts and <code>__callt</code> functions.                         |
| <code>.textf</code>             | Holds <code>__far_func</code> functions.                                                                         |
| <code>.text_unit64kp</code>     | This section is used internally by the linker.                                                                   |
| <code>.wrkseg</code>            | Holds the short address working area; an extra register variable area.                                           |
| <code>.vector</code>            | This section is used internally by the linker.                                                                   |

Table 38: Section summary (Continued)

In addition to the ELF sections used for your application, the tools use a number of other ELF sections for a variety of purposes:

- Sections starting with `.debug` generally contain debug information in the DWARF format

- Sections starting with `.iar.debug` contain supplemental debug information in an IAR format
- The section `.comment` contains the tools and command lines used for building the file
- Sections starting with `.rel` or `.rela` contain ELF relocation information
- The section `.symtab` contains the symbol table for a file
- The section `.strtab` contains the names of the symbol in the symbol table
- The section `.shstrtab` contains the names of the sections.

---

## Descriptions of sections and blocks

This section gives reference information about each section, where the:

- *Description* describes what type of content the section is holding and, where required, how the section is treated by the linker
- *Memory placement* describes memory placement restrictions.

For information about how to allocate sections in memory by modifying the linker configuration file, see *Placing code and data—the linker configuration file*, page 91.

### **.bss**

|                  |                                                                         |
|------------------|-------------------------------------------------------------------------|
| Description      | Holds zero-initialized <code>__near</code> static and global variables. |
| Memory placement | 0xF0000–0xFFE1F in RAM memory.                                          |
| See also         | <i>Memory types</i> , page 66.                                          |

### **.bss.noinit**

|                  |                                                                  |
|------------------|------------------------------------------------------------------|
| Description      | Holds static and global <code>__no_init __near</code> variables. |
| Memory placement | 0xF0000–0xFFE1F in RAM memory.                                   |
| See also         | <i>Memory types</i> , page 66.                                   |

### **.bssf**

|             |                                                                        |
|-------------|------------------------------------------------------------------------|
| Description | Holds zero-initialized <code>__far</code> static and global variables. |
|-------------|------------------------------------------------------------------------|

Memory placement `0x10000–0xFFE1F` in RAM memory.

See also *Memory types*, page 66.

### **.bssf.noinit**

Description Holds static and global `__no_init __far` variables.

Memory placement `0x10000–0xFFE1F` in RAM memory.

See also *Memory types*, page 66.

### **.callt0**

Description Holds call table vectors generated by use of the `__callt` keyword.

Memory placement `0x00080–0x000BF` in ROM memory.

See also `__callt`, page 340.

### **.const**

Description Holds `__near` constant data. This can include constant variables, string and aggregate literals, etc.

Memory placement In a ROM area to be mirrored by hardware to RAM. The location of this area is chip-dependent.

See also *Memory types*, page 66.

### **.constf**

Description Holds `__far` constant data. This can include constant variables, string and aggregate literals, etc.

Memory placement `0x000D0–0xEFFFF` in ROM memory.

See also *Memory types*, page 66.



**.consth**

|                  |                                                                                                                   |
|------------------|-------------------------------------------------------------------------------------------------------------------|
| Description      | Holds <code>__huge</code> constant data. This can include constant variables, string and aggregate literals, etc. |
| Memory placement | 0x000D0–0xEFFFF in ROM memory.                                                                                    |
| See also         | <i>Memory types</i> , page 66.                                                                                    |

**CSTACK**

|                  |                                            |
|------------------|--------------------------------------------|
| Description      | Block that holds the internal data stack.  |
| Memory placement | 0xF0000–0xFFE1F in RAM memory.             |
| See also         | <i>Setting up stack memory</i> , page 109. |

**.data**

|                  |                                                                                                                                                                                                                                                                                                                                            |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds <code>__near</code> static and global initialized variables. In object files, this includes the initial values. When the linker directive <code>initialize</code> is used, a corresponding <code>.near.data_init</code> section is created for each <code>.near.data</code> section, holding the possibly compressed initial values. |
| Memory placement | 0xF0000–0xFFE1F in RAM memory.                                                                                                                                                                                                                                                                                                             |
| See also         | <i>Memory types</i> , page 66.                                                                                                                                                                                                                                                                                                             |

**.data\_init**

|                  |                                                                                                                                                                              |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds the possibly compressed initial values for <code>.data</code> sections. This section is created by the linker if the <code>initialize</code> linker directive is used. |
| Memory placement | 0xF0000–0xFFE1F in RAM memory.                                                                                                                                               |
| See also         | <i>Memory types</i> , page 66.                                                                                                                                               |

## **.dataf**

|                  |                                                                                                                                                                                                                                                                                                                                   |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds <code>__far</code> static and global initialized variables. In object files, this includes the initial values. When the linker directive <code>initialize</code> is used, a corresponding <code>.dataf_init</code> section is created for each <code>.dataf</code> section, holding the possibly compressed initial values. |
| Memory placement | 0x10000–0xFFE1F in RAM memory.                                                                                                                                                                                                                                                                                                    |
| See also         | <i>Memory types</i> , page 66.                                                                                                                                                                                                                                                                                                    |

## **.dataf\_init**

|                  |                                                                                                                                                                               |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds the possibly compressed initial values for <code>.dataf</code> sections. This section is created by the linker if the <code>initialize</code> linker directive is used. |
| Memory placement | 0x10000–0xFFE1F in RAM memory.                                                                                                                                                |
| See also         | <i>Memory types</i> , page 66.                                                                                                                                                |

## **FAR\_HEAP**

|                  |                                                                                                                                                                                                               |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds the heap used for dynamically allocated data in far memory, in other words data allocated by <code>far_malloc</code> and <code>far_free</code> , and in C++, <code>new</code> and <code>delete</code> . |
| Memory placement | 0x10000–0xFFE1F in RAM memory.                                                                                                                                                                                |
| See also         | <i>Setting up heap memory</i> , page 110.                                                                                                                                                                     |

## **.hbss**

|                  |                                                                         |
|------------------|-------------------------------------------------------------------------|
| Description      | Holds zero-initialized <code>__huge</code> static and global variables. |
| Memory placement | 0x10000–0xFFE1F in RAM memory.                                          |
| See also         | <i>Memory types</i> , page 66.                                          |

## **.hbss.noinit**

|             |                                                                  |
|-------------|------------------------------------------------------------------|
| Description | Holds static and global <code>__no_init __huge</code> variables. |
|-------------|------------------------------------------------------------------|

Memory placement 0x10000–0xFFE1F in RAM memory.

See also *Memory types*, page 66.

## **.hdata**

Description Holds `__huge` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize` is used, a corresponding `.hdata_init` section is created for each `.hdata` section, holding the possibly compressed initial values.

Memory placement 0x10000–0xFFE1F in RAM memory.

See also *Memory types*, page 66.

## **.hdata\_init**

Description Holds the possibly compressed initial values for `.hdata` sections. This section is created by the linker if the `initialize` linker directive is used.

Memory placement 0x10000–0xFFE1F in RAM memory.

See also *Memory types*, page 66.

## **HUGE\_HEAP**

Description Holds the heap used for dynamically allocated data in huge memory, in other words data allocated by `huge_malloc` and `huge_free`, and in C++, `new` and `delete`.

Memory placement 0x10000–0xFFE1F in RAM memory.

See also *Setting up heap memory*, page 110.

## **.iar.dynexit**

Description Holds the table of calls to be made at exit.

Memory placement This section can be placed anywhere in ROM memory.

See also *Setting up the atexit limit*, page 110.

## **\_\_iar\_tls.\$\$DATA**

|                  |                                                                                                                                         |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds initial values for TLS variables. This section is created by the linker if the linker option <code>--threaded_lib</code> is used. |
| Memory placement | This section can be placed in ROM anywhere in the 1 Mbyte of memory, but separated into 64-Kbyte sections.                              |
| See also         | <i>Managing a multithreaded environment</i> , page 157                                                                                  |

## **.init\_array**

|                  |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| Description      | Holds pointers to routines to call for initializing one or more C++ objects with static storage duration. |
| Memory placement | 0x000D0–0xEFFFF in ROM memory.                                                                            |

## **.intvec**

|                  |                                                                                                         |
|------------------|---------------------------------------------------------------------------------------------------------|
| Description      | Holds the interrupt vector table generated by the use of the <code>__interrupt</code> extended keyword. |
| Memory placement | 0x00000–0x0007F in ROM memory.                                                                          |

## **NEAR\_HEAP**

|                  |                                                                                                                                                                                                                  |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds the heap used for dynamically allocated data in near memory, in other words data allocated by <code>near_malloc</code> and <code>near_free</code> , and in C++, <code>new</code> and <code>delete</code> . |
| Memory placement | 0xF0000–0xFFE1F in RAM memory.                                                                                                                                                                                   |
| See also         | <i>Setting up heap memory</i> , page 110.                                                                                                                                                                        |

## **.option\_byte**

|                  |                                                                                           |
|------------------|-------------------------------------------------------------------------------------------|
| Description      | Holds the OCD option bytes that are used for configuring the on-chip debugging interface. |
| Memory placement | 0x000C0–0x000C3 in ROM memory.                                                            |

See also *The C-SPY® Debugging Guide for RL78.*

## **.preinit\_array**

**Description** Like `.init_array`, but is used by the library to make some C++ initializations happen before the others.

**Memory placement** `0x000D0–0xEFFFF` in ROM memory.

**See also** `.init_array`, page 444.

## **.sbss**

**Description** Holds zero-initialized `__saddr` static and global variables.

**Memory placement** `0xFFE20–0xFFEDF` in RAM memory.

**See also** *Memory types*, page 66.

## **.sbss.noinit**

**Description** Holds static and global `__no_init __saddr` variables.

**Memory placement** `0xFFE20–0xFFEDF` in RAM memory.

**See also** *Memory types*, page 66.

## **.sdata**

**Description** Holds `__saddr` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize` is used, a corresponding `.sdata_init` section is created for each `.sdata` section, holding the possibly compressed initial values.

**Memory placement** `0xFFE20–0xFFEDF` in RAM memory.

**See also** *Memory types*, page 66.

## **.sdata\_init**

|                  |                                                                                                                                                                               |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds the possibly compressed initial values for <code>.sdata</code> sections. This section is created by the linker if the <code>initialize</code> linker directive is used. |
| Memory placement | 0xFFE20–0xFFEDF in RAM memory.                                                                                                                                                |
| See also         | <i>Memory types</i> , page 66.                                                                                                                                                |

## **.security\_id**

|                  |                                                                                              |
|------------------|----------------------------------------------------------------------------------------------|
| Description      | Holds the Security ID that allows an authentication check before a debug session is started. |
| Memory placement | 0x000C4–0x000CD in ROM memory.                                                               |
| See also         | The <i>C-SPY® Debugging Guide for RL78</i> .                                                 |

## **.switch**

|                  |                                             |
|------------------|---------------------------------------------|
| Description      | Holds switch tables in the Near code model. |
| Memory placement | 0x000D0–0xFFFFF in ROM memory.              |

## **.switchf**

|                  |                                            |
|------------------|--------------------------------------------|
| Description      | Holds switch tables in the Far code model. |
| Memory placement | 0x000D0–0xFFFFF in ROM memory.             |

## **.text**

|                  |                                                                                                                                                                                                                |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description      | The default section for program code. This section holds startup and runtime library code, and code declared with the <code>__near_func</code> , <code>__interrupt</code> , or <code>__callt</code> attribute. |
| Memory placement | 0x000D0–0xFFFFF in ROM memory.                                                                                                                                                                                 |
| See also         | <i>Memory types</i> , page 66.                                                                                                                                                                                 |

## **.textf**

|                  |                                                                                      |
|------------------|--------------------------------------------------------------------------------------|
| Description      | This section holds program code declared with the <code>__far_func</code> attribute. |
| Memory placement | 0x000D0–0xEFFFF in ROM memory.                                                       |
| See also         | <i>Memory types</i> , page 66.                                                       |

## **.wrkseg**

|                  |                                                                                                                                    |
|------------------|------------------------------------------------------------------------------------------------------------------------------------|
| Description      | Holds the short address working area when the <code>--workseg_area</code> option is used. This is an extra register variable area. |
| Memory placement | 0xFFE20–0xFFEDF in RAM memory.                                                                                                     |





# The stack usage control file

- Overview
- Stack usage control directives
- Syntactic components

Before you read this chapter, see *Stack usage analysis*, page 97.

---

## 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`.

**Note:** To comply with the RL78 ABI, the compiler generates assembler labels for C symbols like function names by prefixing an underscore. You must remember to add this extra underscore when you refer to C symbols in any of the stack usage control directives. For example, `main` must be written as `_main`.

### C++ NAMES

When you specify the name of a C++ function in a stack usage control file, you must use the name exactly as used by the linker. Both the number and names of parameters, as well as the names of types must match. However, most non-significant white-space differences are accepted. In particular, you must enclose the name in quote marks because all C++ function names include non-identifier characters.

You can also use wildcards in function names. "`##`" matches any sequence of characters, and "`#?`" matches a single character. This makes it possible to write function names that will match any instantiation of a template function.

Examples:

```
"operator new(unsigned int)"
"std::ostream::flush()"
"operator <<(std::ostream &, char const *)"
"void _Sort<##>(##, ##, ##)"
```

---

## 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 [ category ] : func-spec [ , func-spec... ] ;</code>                                                                                                                                                                                                                                                                                                                     |                                 |
| Parameters  | <i>category</i>                                                                                                                                                                                                                                                                                                                                                                                | See <i>category</i> , page 453  |
|             | <i>func-spec</i>                                                                                                                                                                                                                                                                                                                                                                               | See <i>func-spec</i> , page 453 |
| Description | <p>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.</p> <p>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.</p> |                                 |
| Example     | <code>call graph root [task]: _MyFunc10, _MyFunc11;</code>                                                                                                                                                                                                                                                                                                                                     |                                 |
| See also    | <i>call_graph_root</i> , page 358.                                                                                                                                                                                                                                                                                                                                                             |                                 |

### exclude directive

|             |                                                                                                               |                                 |
|-------------|---------------------------------------------------------------------------------------------------------------|---------------------------------|
| Syntax      | <code>exclude func-spec [ , func-spec... ] ;</code>                                                           |                                 |
| Parameters  | <i>func-spec</i>                                                                                              | See <i>func-spec</i> , page 453 |
| Description | <p>Excludes the specified functions, and call trees originating with them, from stack usage calculations.</p> |                                 |
| Example     | <code>exclude _MyFunc5, _MyFunc6;</code>                                                                      |                                 |

### function directive

|        |                                                                                             |  |
|--------|---------------------------------------------------------------------------------------------|--|
| Syntax | <code>[ override ] function [ category ] func-spec : stack-size [ , call-info... ] ;</code> |  |
|--------|---------------------------------------------------------------------------------------------|--|

|             |                                                                                                                                                                                                                                                                                                                                                                                                          |                                  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| Parameters  | <i>category</i>                                                                                                                                                                                                                                                                                                                                                                                          | See <i>category</i> , page 453   |
|             | <i>func-spec</i>                                                                                                                                                                                                                                                                                                                                                                                         | See <i>func-spec</i> , page 453  |
|             | <i>call-info</i>                                                                                                                                                                                                                                                                                                                                                                                         | See <i>call-info</i> , page 454  |
|             | <i>stack-size</i>                                                                                                                                                                                                                                                                                                                                                                                        | See <i>stack-size</i> , page 454 |
| 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  | <i>func-spec</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | See <i>func-spec</i> , page 453 |
|             | <i>size</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | See <i>size</i> , page 455      |
| Description | <p>Specifies the maximum number of iterations through any of the cycles in the recursion nest of which the function is a member.</p> <p>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.</p> <p>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.</p> |                                 |
| Example     | <pre>max recursion depth _MyFunc12: 10;</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                 |

## no calls from directive

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| Syntax      | <code>no calls from <i>module-spec</i> to <i>func-spec</i> [ , <i>func-spec</i>... ];</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                   |
| Parameters  | <i>func-spec</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | See <i>func-spec</i> , page 453   |
|             | <i>module-spec</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | See <i>module-spec</i> , page 453 |
| Description | <p>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.</p> <p>If there actually is no call to some of these functions, use the <code>no calls from</code> directive to selectively suppress the warning for the specified functions. You can also disable the warning entirely (<code>--diag_suppress</code> or <b>Project&gt;Options&gt;Linker&gt;Diagnostics&gt;Suppress these diagnostics</b>).</p> |                                   |
| Example     | <code>no calls from [file.o] to _MyFunc13, _MyFunc14;</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                   |

## possible calls directive

|             |                                                                                                                                                                                                                                                                                                                                                                                                                          |                                 |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| Syntax      | <code>possible calls <i>calling-func</i> : <i>called-func</i> [ , <i>called-func</i>... ];</code>                                                                                                                                                                                                                                                                                                                        |                                 |
| Parameters  | <i>calling-func</i>                                                                                                                                                                                                                                                                                                                                                                                                      | See <i>func-spec</i> , page 453 |
|             | <i>called-func</i>                                                                                                                                                                                                                                                                                                                                                                                                       | See <i>func-spec</i> , page 453 |
| Description | <p>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.</p> |                                 |
| 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 357.                                                                                                                                                                                                                                                                                                                                                                                                 |                                 |

## Syntactic components

This section describes the syntactical components that can be used by the stack usage control directives.

### ***category***

|             |                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------|
| Syntax      | [ <i>name</i> ]                                                                               |
| Description | A call graph root category. You can use any name you like. Categories are not case-sensitive. |
| Example     | category examples:<br><br>[interrupt]<br>[task]                                               |

### ***func-spec***

|             |                                                                                                                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | [ ? ] <i>name</i> [ <i>module-spec</i> ]                                                                                                                                                                                                     |
| 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 ? suppresses this warning. |
| Example     | <i>func-spec</i> examples:<br><br>_xFun<br>_MyFun [file.o]<br>?"fun1(int) "                                                                                                                                                                  |

### ***module-spec***

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | [name [ ( <i>name</i> ) ]]                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Description | 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: <ul style="list-style-type: none"> <li>• The complete path of the file ("D:\C1\test\file.o")</li> <li>• As many path elements as are needed at the end of the path ("test\file.o")</li> <li>• Some path elements at the start of the path, followed by "...", followed by some path elements at the end ("D:\...\file.o").</li> </ul> |

**Note:** When using multi-file compilation (`--mfc`), multiple files are compiled into a single module, named after the first file.

Example `module-spec` examples:

```
[file.o]
[file.o(lib.a)]
["D:\C1\test\file.o"]
```

## ***name***

Description A name can be either an identifier or a quoted string.

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.

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 " .

Example `name` examples:

```
_MyFun
file.o
"file-1.o"
```

## ***call-info***

Syntax `calls func-spec [ , func-spec... ] [ : stack-size ]`

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`  
`( [ stack ] size )`

Description Specifies the size of a stack frame. A stack may not be specified more than once.

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
```





# IAR utilities

- The IAR Archive Tool—`iarchive`—creates and manipulates a library (an archive) of several ELF object files
- The IAR ELF Tool—`ielftool`—performs various transformations on an ELF executable image (such as fill, checksum, format conversions, etc)
- The IAR ELF Dumper—`ielfdump`—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—`iobjmanip`—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—`ismexport`—exports absolute symbols from a ROM image file, so that they can be used when you link an add-on application.
- Descriptions of options—detailed reference information about each command line option available for the different utilities.

---

## The IAR Archive Tool—`iarchive`

The IAR Archive Tool, `iarchive`, can create a library (an archive) file from several ELF object files. You can also use `iarchive` to manipulate ELF libraries.

A library file contains several relocatable ELF object modules, each of which can be independently used by a linker. In contrast with object modules specified directly to the linker, each module in a library is only included if it is needed.

For information about how to build a library in the IDE, see the *IDE Project Management and Building Guide for RL78*.

### INVOCATION SYNTAX

The invocation syntax for the archive builder is:

```
iarchive parameters
```

## Parameters

The parameters are:

| Parameter                                    | Description                                                                                                                          |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>command</i>                               | Command line options that define an operation to be performed. Such an option must be specified before the name of the library file. |
| <i>libraryfile</i>                           | The library file to be operated on.                                                                                                  |
| <i>objectfile1</i> ...<br><i>objectfileN</i> | The object file(s) that the specified command operates on.                                                                           |
| <i>options</i>                               | Command line options that define actions to be performed. These options can be placed anywhere on the command line.                  |

Table 39: *iarchive* parameters

## Examples

This example creates a library file called `mylibrary.a` from the source object files `module1.o`, `module2.o`, and `module3.o`:

```
iarchive mylibrary.a module1.o module2.o module3.o.
```

This example lists the contents of `mylibrary.a`:

```
iarchive --toc mylibrary.a
```

This example replaces `module3.o` in the library with the content in the `module3.o` file and appends `module4.o` to `mylibrary.a`:

```
iarchive --replace mylibrary.a module3.o module4.o
```

## SUMMARY OF IARCHIVE COMMANDS

This table summarizes the `iarchive` commands:

| Command line option        | Description                                                 |
|----------------------------|-------------------------------------------------------------|
| <code>--create</code>      | Creates a library that contains the listed object files.    |
| <code>--delete, -d</code>  | Deletes the listed object files from the library.           |
| <code>--extract, -x</code> | Extracts the listed object files from the library.          |
| <code>--replace, -r</code> | Replaces or appends the listed object files to the library. |
| <code>--symbols</code>     | Lists all symbols defined by files in the library.          |
| <code>--toc, -t</code>     | Lists all files in the library.                             |

Table 40: *iarchive* commands summary

For more information, see *Descriptions of options*, page 473.

## SUMMARY OF IARCHIVE OPTIONS

This table summarizes the `iarchive` command line options:

| Command line option         | Description                                             |
|-----------------------------|---------------------------------------------------------|
| <code>-f</code>             | Extends the command line.                               |
| <code>--f</code>            | Extends the command line, optionally with a dependency. |
| <code>--no_bom</code>       | Omits the byte order mark from UTF-8 output files.      |
| <code>--output, -o</code>   | Specifies the library file.                             |
| <code>--text_out</code>     | Specifies the encoding for text output files.           |
| <code>--utf8_text_in</code> | Uses the UTF-8 encoding for text input files.           |
| <code>--verbose, -V</code>  | Reports all performed operations.                       |
| <code>--version</code>      | Sends tool output to the console and then exits.        |
| <code>--vtoc</code>         | Produces a verbose list of files in the library.        |

Table 41: *iarchive* options summary

For more information, see *Descriptions of options*, page 473.

## DIAGNOSTIC MESSAGES

This section lists the messages produced by `iarchive`:

### **La001: could not open file *filename***

`iarchive` failed to open an object file.

### **La002: illegal path *pathname***

The path *pathname* is not a valid path.

### **La006: too many parameters to *cmd* command**

A list of object modules was specified as parameters to a command that only accepts a single library file.

### **La007: too few parameters to *cmd* command**

A command that takes a list of object modules was issued without the expected modules.

### **La008: *lib* is not a library file**

The library file did not pass a basic syntax check. Most likely the file is not the intended library file.

**La009: *lib* has no symbol table**

The library file does not contain the expected symbol information. The reason might be that the file is not the intended library file, or that it does not contain any ELF object modules.

**La010: no library parameter given**

The tool could not identify which library file to operate on. The reason might be that a library file has not been specified.

**La011: file *file* already exists**

The file could not be created because a file with the same name already exists.

**La013: file confusions, *lib* given as both library and object**

The library file was also mentioned in the list of object modules.

**La014: module *module* not present in archive *lib***

The specified object module could not be found in the archive.

**La015: internal error**

The invocation triggered an unexpected error in `iarchive`.

**Ms003: could not open file *filename* for writing**

`iarchive` failed to open the archive file for writing. Make sure that it is not write protected.

**Ms004: problem writing to file *filename***

An error occurred while writing to file `filename`. A possible reason for this is that the volume is full.

**Ms005: problem closing file *filename***

An error occurred while closing the file `filename`.

---

## The IAR ELF Tool—ielftool

The IAR ELF Tool, `ielftool`, can generate a checksum on specific ranges of memories. This checksum can be compared with a checksum calculated on your application.

The source code for `ielftool` and a Microsoft VisualStudio template project are available in the `r178\src\elfutils` directory. If you have specific requirements for how the checksum should be generated or requirements for format conversion, you can modify the source code accordingly.

## INVOCATION SYNTAX

The invocation syntax for the IAR ELF Tool is:

```
ielftool [options] inputfile outputfile [options]
```

The `ielftool` tool will first process all the fill options, then it will process all the checksum options (from left to right).

## Parameters

The parameters are:

| Parameter         | Description                                                                                                                     |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <i>inputfile</i>  | An absolute ELF executable image produced by the ILINK linker.                                                                  |
| <i>options</i>    | Any of the available command line options, see <i>Summary of ielftool options</i> , page 461.                                   |
| <i>outputfile</i> | An absolute ELF executable image, or if one of the relevant command line options is specified, an image file in another format. |

Table 42: *ielftool* parameters

See also *Rules for specifying a filename or directory as parameters*, page 250.

## Example

This example fills a memory range with `0xFF` and then calculates a checksum on the same range:

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
 --checksum __checksum:4,crc32;0-0xFF
```

## SUMMARY OF IELFTOOL OPTIONS

This table summarizes the `ielftool` command line options:

| Command line option      | Description                                       |
|--------------------------|---------------------------------------------------|
| <code>--bin</code>       | Sets the format of the output file to raw binary. |
| <code>--bin-multi</code> | Produces output to multiple raw binary files.     |
| <code>--checksum</code>  | Generates a checksum.                             |
| <code>--fill</code>      | Specifies fill requirements.                      |

Table 43: *ielftool* options summary

| Command line option          | Description                                                                  |
|------------------------------|------------------------------------------------------------------------------|
| <code>--front_headers</code> | Outputs headers in the beginning of the file.                                |
| <code>--ihex</code>          | Sets the format of the output file to 32-bit linear Intel Extended hex.      |
| <code>--offset</code>        | Adds (or subtracts) an offset to all addresses in the generated output file. |
| <code>--parity</code>        | Generates parity bits.                                                       |
| <code>--self_reloc</code>    | Not for general use.                                                         |
| <code>--silent</code>        | Sets silent operation.                                                       |
| <code>--simple</code>        | Sets the format of the output file to Simple-code.                           |
| <code>--simple-ne</code>     | As <code>--simple</code> , but without an entry record.                      |
| <code>--srec</code>          | Sets the format of the output file to Motorola S-records.                    |
| <code>--srec-len</code>      | Restricts the number of data bytes in each S-record.                         |
| <code>--srec-s3only</code>   | Restricts the S-record output to contain only a subset of records.           |
| <code>--strip</code>         | Removes debug information.                                                   |
| <code>--titxt</code>         | Sets the format of the output file to Texas Instruments TI-TXT.              |
| <code>--verbose, -V</code>   | Prints all performed operations.                                             |
| <code>--version</code>       | Sends tool output to the console and then exits.                             |

Table 43: *ielftool options summary (Continued)*

For more information, see *Descriptions of options*, page 473.

## The IAR ELF Dumper—ielfdump

The IAR ELF Dumper for RL78, `ielfdumprl78`, can be used for creating a text representation of the contents of a relocatable or absolute ELF file.

`ielfdumprl78` can be used in one of three ways:

- To produce a listing of the general properties of the input file and the ELF segments and ELF sections it contains. This is the default behavior when no command line options are used.
- To also include a textual representation of the contents of each ELF section in the input file. To specify this behavior, use the command line option `--all`.
- To produce a textual representation of selected ELF sections from the input file. To specify this behavior, use the command line option `--section`.

## INVOCATION SYNTAX

The invocation syntax for `ielfdumppr178` is:

```
ielfdumppr178 input_file [output_file]
```

**Note:** `ielfdumppr178` is a command line tool which is not primarily intended to be used in the IDE.

## Parameters

The parameters are:

| Parameter                | Description                                                                                                                                     |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>input_file</code>  | An ELF relocatable or executable file to use as input.                                                                                          |
| <code>output_file</code> | A file or directory where the output is emitted. If absent and no <code>--output</code> option is specified, output is directed to the console. |

Table 44: `ielfdumppr178` parameters

See also *Rules for specifying a filename or directory as parameters*, page 250.

## SUMMARY OF IELFDUMP OPTIONS

This table summarizes the `ielfdumppr178` command line options:

| Command line option           | Description                                                                   |
|-------------------------------|-------------------------------------------------------------------------------|
| <code>--a</code>              | Generates output for all sections except string table sections.               |
| <code>--all</code>            | Generates output for all input sections regardless of their names or numbers. |
| <code>--code</code>           | Dumps all sections that contain executable code.                              |
| <code>--disasm_data</code>    | Dumps data sections as code sections.                                         |
| <code>-f</code>               | Extends the command line.                                                     |
| <code>--f</code>              | Extends the command line, optionally with a dependency.                       |
| <code>--no_bom</code>         | Omits the Byte Order Mark from UTF-8 output files.                            |
| <code>--no_header</code>      | Suppresses production of a list header in the output.                         |
| <code>--no_rel_section</code> | Suppresses dumping of <code>.rel/.rela</code> sections.                       |
| <code>--no_strtab</code>      | Suppresses dumping of string table sections.                                  |
| <code>--no_utf8_in</code>     | Do not assume UTF-8 for non-IAR ELF files.                                    |
| <code>--output, -o</code>     | Specifies an output file.                                                     |
| <code>--range</code>          | Disassembles only addresses in the specified range.                           |

Table 45: `ielfdumppr178` options summary

| Command line option                             | Description                                                                                                                                         |
|-------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--raw</code>                              | Uses the generic hexadecimal/ASCII output format for the contents of any selected section, instead of any dedicated output format for that section. |
| <code>--section, -s</code>                      | Generates output for selected input sections.                                                                                                       |
| <code>--segment, -g</code>                      | Generates output for segments with specified numbers.                                                                                               |
| <code>--source</code>                           | Includes source with disassembled code in executable files.                                                                                         |
| <code>--text_out</code>                         | Specifies the encoding for text output files.                                                                                                       |
| <code>--use_full_std_t<br/>emplate_names</code> | Uses full short full names for some Standard C++ templates.                                                                                         |
| <code>--utf8_text_in</code>                     | Uses the UTF-8 encoding for text input files.                                                                                                       |
| <code>--version</code>                          | Sends tool output to the console and then exits.                                                                                                    |

Table 45: `ielddump178` options summary (Continued)

For more information, see *Descriptions of options*, page 473.

## The IAR ELF Object Tool—iobjmanip

Use the IAR ELF Object Tool, `iobjmanip`, to perform low-level manipulation of ELF object files.

### INVOCATION SYNTAX

The invocation syntax for the IAR ELF Object Tool is:

```
iobjmanip options inputfile outputfile
```

### Parameters

The parameters are:

| Parameter               | Description                                                                                                                                                        |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>options</code>    | Command line options that define actions to be performed. These options can be placed anywhere on the command line. At least one of the options must be specified. |
| <code>inputfile</code>  | A relocatable ELF object file.                                                                                                                                     |
| <code>outputfile</code> | A relocatable ELF object file with all the requested operations applied.                                                                                           |

Table 46: `iobjmanip` parameters

See also *Rules for specifying a filename or directory as parameters*, page 250.



## Examples

This example renames the section `.example` in `input.o` to `.example2` and stores the result in `output.o`:

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

## SUMMARY OF IOBJMANIP OPTIONS

This table summarizes the `iobjmanip` options:

| Command line option             | Description                                             |
|---------------------------------|---------------------------------------------------------|
| <code>-f</code>                 | Extends the command line.                               |
| <code>--f</code>                | Extends the command line, optionally with a dependency. |
| <code>--no_bom</code>           | Omits the Byte Order Mark from UTF-8 output files.      |
| <code>--remove_file_path</code> | Removes path information from the file symbol.          |
| <code>--remove_section</code>   | Removes one or more section.                            |
| <code>--rename_section</code>   | Renames a section.                                      |
| <code>--rename_symbol</code>    | Renames a symbol.                                       |
| <code>--strip</code>            | Removes debug information.                              |
| <code>--text_out</code>         | Specifies the encoding for text output files.           |
| <code>--utf8_text_in</code>     | Uses the UTF-8 encoding for text input files.           |
| <code>--version</code>          | Sends tool output to the console and then exits.        |

Table 47: *iobjmanip* options summary

For more information, see *Descriptions of options*, page 473.

## DIAGNOSTIC MESSAGES

This section lists the messages produced by `iobjmanip`:

### Lm001: No operation given

None of the command line parameters specified an operation to perform.

### Lm002: Expected *nr* parameters but got *nr*

Too few or too many parameters. Check invocation syntax for `iobjmanip` and for the used command line options.

### Lm003: Invalid section/symbol renaming pattern *pattern*

The pattern does not define a valid renaming operation.

**Lm004: Could not open file *filename***

iobjmanip failed to open the input file.

**Lm005: ELF format error *msg***

The input file is not a valid ELF object file.

**Lm006: Unsupported section type *nr***

The object file contains a section that iobjmanip cannot handle. This section will be ignored when generating the output file.

**Lm007: Unknown section type *nr***

iobjmanip encountered an unrecognized section. iobjmanip will try to copy the content as is.

**Lm008: Symbol *symbol* has unsupported format**

iobjmanip encountered a symbol that cannot be handled. iobjmanip will ignore this symbol when generating the output file.

**Lm009: Group type *nr* not supported**

iobjmanip only supports groups of type GRP\_COMDAT. If any other group type is encountered, the result is undefined.

**Lm010: Unsupported ELF feature in *file*: *msg***

The input file uses a feature that iobjmanip does not support.

**Lm011: Unsupported ELF file type**

The input file is not a relocatable object file.

**Lm012: Ambiguous rename for section/symbol name (*alt1* and *alt2*)**

An ambiguity was detected while renaming a section or symbol. One of the alternatives will be used.

**Lm013: Section *name* removed due to transitive dependency on *name***

A section was removed as it depends on an explicitly removed section.

**Lm014: File has no section with index *nr***

A section index, used as a parameter to `--remove_section` or `--rename_section`, did not refer to a section in the input file.

**Ms003: could not open file *filename* for writing**

`iobjmanip` failed to open the output file for writing. Make sure that it is not write protected.

**Ms004: problem writing to file *filename***

An error occurred while writing to file *filename*. A possible reason for this is that the volume is full.

**Ms005: problem closing file *filename***

An error occurred while closing the file *filename*.

---

## The IAR Absolute Symbol Exporter—`ismexport`

The IAR Absolute Symbol Exporter, `ismexport`, can export absolute symbols from a ROM image file, so that they can be used when you link an add-on application.

To keep symbols from your symbols file in your final application, the symbols must be referred to, either from your source code or by using the linker option `--keep`.

**INVOCATION SYNTAX**

The invocation syntax for the IAR Absolute Symbol Exporter is:

```
ismexport [options] inputfile outputfile [options]
```

**Parameters**

The parameters are:

| Parameter        | Description                                                                                    |
|------------------|------------------------------------------------------------------------------------------------|
| <i>inputfile</i> | A ROM image in the form of an executable ELF file (output from linking).                       |
| <i>options</i>   | Any of the available command line options, see <i>Summary of ismexport options</i> , page 468. |

Table 48: `ismexport` parameters

| Parameter               | Description                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>outputfile</code> | A relocatable ELF file that can be used as input to linking, and which contains all or a selection of the absolute symbols in the input file. The output file contains only the symbols, not the actual code or data sections. A steering file can be used for controlling which symbols are included, and if desired, for also renaming some of the symbols. |

Table 48: `isymexport` parameters (Continued)

See also *Rules for specifying a filename or directory as parameters*, page 250.



In the IDE, to add the export of library symbols, choose **Project>Options>Build Actions** and specify your command line in the **Post-build command line** text field, for example:

```
$TOOLKIT_DIR$\bin\isymexport.exe "$TARGET_PATH$"
"$PROJ_DIR$\const_lib.symbols"
```

## SUMMARY OF ISYMEXPORT OPTIONS

This table summarizes the `isymexport` command line options:

| Command line option                | Description                                                                                             |
|------------------------------------|---------------------------------------------------------------------------------------------------------|
| <code>--edit</code>                | Specifies a steering file.                                                                              |
| <code>--export_locals</code>       | Exports local symbols.                                                                                  |
| <code>-f</code>                    | Extends the command line.                                                                               |
| <code>--f</code>                   | Extends the command line, optionally with a dependency.                                                 |
| <code>--generate_vfe_header</code> | Declares that the image does not contain any virtual function calls to potentially discarded functions. |
| <code>--no_bom</code>              | Omits the Byte Order Mark from UTF-8 output files.                                                      |
| <code>--ram_reserve_ranges</code>  | Generates symbols for the areas in RAM that the image uses.                                             |
| <code>--reserve_ranges</code>      | Generates symbols to reserve the areas in ROM and RAM that the image uses.                              |
| <code>--show_entry_as</code>       | Exports the entry point of the application with the given name.                                         |
| <code>--text_out</code>            | Specifies the encoding for text output files.                                                           |
| <code>--utf8_text_in</code>        | Uses the UTF-8 encoding for text input files.                                                           |
| <code>--version</code>             | Sends tool output to the console and then exits.                                                        |

Table 49: `isymexport` options summary

For more information, see *Descriptions of options*, page 473.

## STEERING FILES

A steering file can be used for controlling which symbols are included, and if desired, for also renaming some of the symbols. In the file, you can use `show` and `hide` directives to select which public symbols from the input file that are to be included in the output file. `rename` directives can be used for changing the names of symbols in the input file.

When you use a steering file, only actively exported symbols will be available in the output file. Therefore, a steering file without `show` directives will generate an output file without symbols.

### Syntax

The following syntax rules apply:

- Each directive is specified on a separate line.
- C comments (`/*...*/`) and C++ comments (`//...`) can be used.
- Patterns can contain wildcard characters that match more than one possible character in a symbol name.
- The `*` character matches any sequence of zero or more characters in a symbol name.
- The `?` character matches any single character in a symbol name.

### Example

```
rename xxx_* as YYY_* /*Change symbol prefix from xxx_ to YYY_ */
show YYY_* /* Export all symbols from YYY package */
hide *_internal /* But do not export internal symbols */
show zzz? /* Export zzza, but not zzzaaa */
hide zzzx /* But do not export zzzx */
```

## Hide directive

|             |                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>hide <i>pattern</i></code>                                                                                                                        |
| Parameters  | <code><i>pattern</i></code> A pattern to match against a symbol name.                                                                                   |
| Description | A symbol with a name that matches the pattern will not be included in the output file unless this is overridden by a later <code>show</code> directive. |
| Example     | <pre>/* Do not include public symbols ending in _sys. */ hide *_sys</pre>                                                                               |

## Rename directive

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>rename <i>pattern1</i> as <i>pattern2</i></code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Parameters  | <p><i>pattern1</i>      A pattern used for finding symbols to be renamed. The pattern can contain no more than one * or ? wildcard character.</p> <p><i>pattern2</i>      A pattern used for the new name for a symbol. If the pattern contains a wildcard character, it must be of the same kind as in <i>pattern1</i>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Description | <p>Use this directive to rename symbols from the output file to the input file. No exported symbol is allowed to match more than one <code>rename</code> pattern.</p> <p><code>rename</code> directives can be placed anywhere in the steering file, but they are executed before any <code>show</code> and <code>hide</code> directives. Therefore, if a symbol will be renamed, all <code>show</code> and <code>hide</code> directives in the steering file must refer to the new name.</p> <p>If the name of a symbol matches a <i>pattern1</i> pattern that contains no wildcard characters, the symbol will be renamed <i>pattern2</i> in the output file.</p> <p>If the name of a symbol matches a <i>pattern1</i> pattern that contains a wildcard character, the symbol will be renamed <i>pattern2</i> in the output file, with part of the name matching the wildcard character preserved.</p> |
| Example     | <pre>/* xxx_start will be renamed Y_start_X in the output file,    xxx_stop will be renamed Y_stop_X in the output file. */ rename xxx_* as Y*_X</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

## Show directive

|             |                                                                                                                                                     |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>show <i>pattern</i></code>                                                                                                                    |
| Parameters  | <i>pattern</i> A pattern to match against a symbol name.                                                                                            |
| Description | A symbol with a name that matches the pattern will be included in the output file unless this is overridden by a later <code>hide</code> directive. |
| Example     | <pre>/* Include all public symbols ending in _pub. */ show *_pub</pre>                                                                              |

## Show-root directive

|             |                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>show-root pattern</code>                                                                                                                                                                                                                                                                                                                                                            |
| Parameters  | <code>pattern</code> A pattern to match against a symbol name.                                                                                                                                                                                                                                                                                                                            |
| Description | <p>A symbol with a name that matches the pattern will be included in the output file, marked as <code>root</code>, unless this is overridden by a later <code>hide</code> directive.</p> <p>When linking with the module produced by <code>isymexport</code>, the symbol will be included in the final executable file, even if no references to the symbol are present in the build.</p> |
| Example     | <pre>/* Export myVar making sure that it is included when linking */ show-root myVar</pre>                                                                                                                                                                                                                                                                                                |

## Show-weak directive

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>show-weak pattern</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Parameters  | <code>pattern</code> A pattern to match against a symbol name.                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Description | <p>A symbol with a name that matches the pattern will be included in the output file as a weak symbol unless this is overridden by a later <code>hide</code> directive.</p> <p>When linking, no error will be reported if the new code contains a definition for a symbol with the same name as the exported symbol.</p> <p><b>Note:</b> Any internal references in the <code>isymexport</code> input file are already resolved and cannot be affected by the presence of definitions in the new code.</p> |
| Example     | <pre>/* Export myFunc as a weak definition */ show-weak myFunc</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## DIAGNOSTIC MESSAGES

This section lists the messages produced by `isymexport`:

### **Es001: could not open file *filename***

`isymexport` failed to open the specified file.

**Es002: illegal path *pathname***

The path *pathname* is not a valid path.

**Es003: format error: *message***

A problem occurred while reading the input file.

**Es004: no input file**

No input file was specified.

**Es005: no output file**

An input file, but no output file was specified.

**Es006: too many input files**

More than two files were specified.

**Es007: input file is not an ELF executable**

The input file is not an ELF executable file.

**Es008: unknown directive: *directive***

The specified directive in the steering file is not recognized.

**Es009: unexpected end of file**

The steering file ended when more input was required.

**Es010: unexpected end of line**

A line in the steering file ended before the directive was complete.

**Es011: unexpected text after end of directive**

There is more text on the same line after the end of a steering file directive.

**Es012: expected text**

The specified text was not present in the steering file, but must be present for the directive to be correct.



**Es013: pattern can contain at most one \* or ?**

Each pattern in the current directive can contain at most one \* or one ? wildcard character.

**Es014: rename patterns have different wildcards**

Both patterns in the current directive must contain exactly the same kind of wildcard. That is, both must either contain:

- No wildcards
- Exactly one \*
- Exactly one ?

This error occurs if the patterns are not the same in this regard.

**Es015: ambiguous pattern match: *symbol* matches more than one rename pattern**

A symbol in the input file matches more than one `rename` pattern.

**Es016: the entry point symbol is already exported**

The option `--show_entry_as` was used with a name that already exists in the input file.

---

## Descriptions of options

This section gives detailed reference information about each command line option available for the different utilities.

**--a**

|              |                                                                    |
|--------------|--------------------------------------------------------------------|
| Syntax       | <code>--a</code>                                                   |
| For use with | <code>ieifdump178</code>                                           |
| Description  | Use this option as a shortcut for <code>--all --no_strtab</code> . |



This option is not available in the IDE.

## --all

|              |                                                                                                                                                                                                                                                                                                                                              |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | --all                                                                                                                                                                                                                                                                                                                                        |
| For use with | ielfdumprl78                                                                                                                                                                                                                                                                                                                                 |
| Description  | Use this option to include the contents of all ELF sections in the output, in addition to the general properties of the input file. Sections are output in index order, except that each relocation section is output immediately after the section it holds relocations for.<br>By default, no section contents are included in the output. |



This option is not available in the IDE.

## --bin

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | --bin[= <i>range</i> ]                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Parameters   | <i>range</i> The address range content to include in the output file. The address range can be specified using literals, or by using symbols present in the ELF file. Examples:<br>"0x8000-0x8FFF", "START-END"                                                                                                                                                                                                                                                                           |
| For use with | ielftool                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Description  | Sets the format of the output file to raw binary, a binary format that includes only the raw bytes, with no address information. If no range is specified, the output file will include all the bytes from the lowest address for which there is content in the ELF file to the highest address for which there is content. If a range is specified, only bytes from that range are included. Note that in both cases, any gaps for which there is no content will be generated as zeros. |



To set related options, choose:

**Project>Options>Output converter**

## --bin-multi

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                                                                       |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--bin-multi[=<i>range</i>;<i>range</i>...]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                       |
| Parameters   | <i>range</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | An address range to produce an output file for. An address range can be specified using literals, or by using symbols present in the ELF file. Examples: "0x8000-0x8FFF", "START-END" |
| For use with | ielftool                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                       |
| Description  | Use this option to produce one or more raw binary output files. If no ranges are specified, a raw binary output file is generated for each range for which there is content in the ELF file. If ranges are specified, a raw binary output file is generated for each range specified. In each case, the name of each output file will include the start address of its range. For example, if the output file is specified as <code>out.bin</code> and the ranges <code>0x0-0x1F</code> and <code>0x8000-0x8147</code> are output, there will be two files, named <code>out-0x0.bin</code> and <code>out-0x8000.bin</code> . |                                                                                                                                                                                       |



This option is not available in the IDE.

## --checksum

|            |                                                                                                                                                                                                          |                                                                                                                                      |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>--checksum {<i>symbol</i>[+<i>offset</i>]   <i>address</i>}:<i>size</i>,<br/><i>algorithm</i>[: [1 2] [a m z] [L W] [x r] [R] [o] [i p]]<br/>[, <i>start</i>];<i>range</i>;<i>range</i>...]</code> |                                                                                                                                      |
| Parameters | <i>symbol</i>                                                                                                                                                                                            | The name of the symbol where the checksum value should be stored. Note that it must exist in the symbol table in the input ELF file. |
|            | <i>offset</i>                                                                                                                                                                                            | An offset to the symbol.                                                                                                             |
|            | <i>address</i>                                                                                                                                                                                           | The absolute address where the checksum value should be stored.                                                                      |
|            | <i>size</i>                                                                                                                                                                                              | The number of bytes in the checksum—1, 2, or 4. The number cannot be larger than the size of the checksum symbol.                    |

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>algorithm</i> | <p>The checksum algorithm used. Choose between:</p> <p><i>sum</i>, a byte-wise calculated arithmetic sum. The result is truncated to 8 bits.</p> <p><i>sum8wide</i>, a byte-wise calculated arithmetic sum. The result is truncated to the size of the symbol.</p> <p><i>sum32</i>, a word-wise (32 bits) calculated arithmetic sum.</p> <p><i>crc16</i>, CRC16 (generating polynomial 0x1021); used by default.</p> <p><i>crc32</i>, CRC32 (generating polynomial 0x04C11DB7).</p> <p><i>crc64iso</i>, CRC64iso (generating polynomial 0x1B).</p> <p><i>crc64ecma</i>, CRC64ECMA (generating polynomial 0x42F0E1EBA9EA3693).</p> <p><i>crc=n</i>, CRC with a generating polynomial of <i>n</i>.</p> |
| 1 2              | <p>If specified, choose between:</p> <p>1, specifies one's complement.</p> <p>2, specifies two's complement.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| a m z            | <p>Reverses the order of the bits for the checksum. Choose between:</p> <p><i>a</i>, reverses the input bytes (but nothing else).</p> <p><i>m</i>, reverses the input bytes and the final checksum.</p> <p><i>z</i>, reverses the final checksum (but nothing else).</p> <p>Note that using <i>a</i> and <i>z</i> in combination has the same effect as <i>m</i>.</p>                                                                                                                                                                                                                                                                                                                                |

$L|W$ 

Specifies the size of the unit for which a checksum should be calculated. Choose between:

$L$ , calculates a checksum on 32 bits in every iteration

$w$ , calculates a checksum on 16 bits in every iteration.

If you do not specify a unit size, 8 bits will be used by default.

The input byte sequence will be processed as:

- 8-bit checksum unit size—byte0, byte1, byte2, byte3, etc.
- 16-bit checksum unit size—byte1, byte0, byte3, byte2, etc.
- 32-bit checksum unit size—byte3, byte2, byte1, byte0, byte7, byte6, byte5, byte4, etc.

**Note:** The checksum unit size only affects the order in which the input byte sequence is processed. It does not affect the size of the checksum symbol, the polynomial, the initial value, the width of the processor's address bus, etc.

Most software CRC implementations use a checksum unit size of 1 byte (8 bits). The  $L$  and  $w$  parameters are almost exclusively used when a software CRC implementation has to match the checksum computed by the hardware CRC implementation. If you are not trying to cooperate with a hardware CRC implementation, the  $L$  and  $w$  parameter will simply compute a different checksum, because it processes the input byte sequence in a different order.


 $x$ 

Reverses the byte order of the checksum. This only affects the checksum value.

 $r$ 

Reverses the byte order of the input data. This has no effect unless the number of bits per iteration has been set using the  $L$  or  $w$  parameters.

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>R</code>     | <p>Traverses the checksum range(s) in reverse order.</p> <p>If the range is, for example, <code>0x100-0xFFF;0x2000-0x2FFF</code>, the checksum calculation will normally start on <code>0x100</code> and then calculate every byte up to and including <code>0xFFF</code>, followed by calculating the byte on <code>0x2000</code> and continue to <code>0x2FFF</code>.</p> <p>Using the <code>R</code> parameter, the calculation instead starts on <code>0x2FFF</code> and continues by calculating every byte down to <code>0x2000</code>, then from <code>0xFFF</code> down to and including <code>0x100</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>o</code>     | <p>Outputs the Rocksoft model specification for the checksum.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>i p</code>   | <p>Use either <code>i</code> or <code>p</code>, if the <code>start</code> value is bigger than 0. Choose between:</p> <p><code>i</code>, initializes the checksum value with the start value.</p> <p><code>p</code>, prefixes the input data with a word of size <code>size</code> that contains the <code>start</code> value.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>start</code> | <p>By default, the initial value of the checksum is 0. If necessary, use <code>start</code> to supply a different initial value. If not 0, then either <code>i</code> or <code>p</code> must be specified.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>range</code> | <p><code>range</code> is one or more memory ranges for which the checksum will be calculated. Hexadecimal and decimal notation is allowed, for example, <code>0x8002-0x8FFF</code>. The memory range(s) can also be expressed as:</p> <ul style="list-style-type: none"> <li>● Symbols that are present in ELF file can be used in the range description, for example, <code>__checksum_begin-__checksum_end</code>.</li> <li>● One or more block names where each block is placed inside a pair of curly braces, {}, like <code>{MY_BLOCK}</code>. A block that is used in this manner must be specified in the linker configuration file and must contain only read-only content. See <i>define block directive</i>, page 415.</li> </ul> <p>It is typically advisable to use symbols or blocks if the memory range can change. If you use explicit addresses, for example, <code>0x8000-0x8347</code>, and the code then changes, you need to update the end address to the new value. If you instead use <code>{CODE}</code> or a symbol located at the end of the code, you do not need to update the <code>--checksum</code> command.</p> |

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| For use with | ielftool                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Description  | <p>Use this option to calculate a checksum with the specified algorithm for the specified ranges. If you have an external definition for the checksum—for example, a hardware CRC implementation—use the appropriate parameters to the <code>--checksum</code> option to match the external design. In this case, learn more about that design in the hardware documentation. The checksum will then replace the original value in <i>symbol</i>. A new absolute symbol will be generated, with the <i>symbol</i> name suffixed with <code>_value</code> containing the calculated checksum. This symbol can be used for accessing the checksum value later when needed, for example, during debugging.</p> <p>If the <code>--checksum</code> option is used more than once on the command line, the options are evaluated from left to right. If a checksum is calculated for a <i>symbol</i> that is specified in a later evaluated <code>--checksum</code> option, an error is issued.</p>                                                                                                                                         |
| Example      | <p>This example shows how to use the <code>crc16</code> algorithm with the start value 0 over the address range <code>0x8000-0x8FFF</code>:</p> <pre>ielftool --checksum=__checksum:2,crc16;0x8000-0x8FFF sourceFile.out destinationFile.out</pre> <p>The input data <i>i</i> read from <code>sourceFile.out</code>, and the resulting checksum value of size 2 bytes will be stored at the symbol <code>_checksum</code>. The modified ELF file is saved as <code>destinationFile.out</code> leaving <code>sourceFile.out</code> untouched.</p> <p>In the next example, a symbol is used for specifying the start of the range:</p> <pre>ielftool --checksum=__checksum:2,crc16;__checksum_begin-0x8FFF sourceFile.out destinationFile.out</pre> <p>If <code>BLOCK1</code> occupies <code>0x4000-0x4337</code> and <code>BLOCK2</code> occupies <code>0x8000-0x87FF</code>, this example will compute the checksum for the bytes on <code>0x4000</code> to <code>0x4337</code> and from <code>0x8000</code> to <code>0x87FF</code>:</p> <pre>ielftool --checksum __checksum:2,crc16;{BLOCK1};{BLOCK2} BlxTest.out BlxTest2.out</pre> |
| See also     | <p><i>Checksum calculation for verifying image integrity</i>, page 209</p> <p> To set related options, choose:<br/><b>Project&gt;Options&gt;Linker&gt;Checksum</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

## --code

|              |                                                                                                                                        |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--code</code>                                                                                                                    |
| For use with | <code>ielfdumppr178</code>                                                                                                             |
| Description  | Use this option to dump all sections that contain executable code—sections with the ELF section attribute <code>SHF_EXECINSTR</code> . |



This option is not available in the IDE.

## --create

|              |                                                                                                                                                                                                                                                                                           |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--create libraryfile objectfile1 ... objectfileN</code>                                                                                                                                                                                                                             |
| Parameters   | <p><code>libraryfile</code>      The library file that the command operates on. See <i>Rules for specifying a filename or directory as parameters</i>, page 250.</p> <p><code>objectfile1 ...</code>    The object file(s) to build the library from.</p> <p><code>objectfileN</code></p> |
| For use with | <code>iarchive</code>                                                                                                                                                                                                                                                                     |
| Description  | Use this command to build a new library from a set of object files (modules). The object files are added to the library in the exact order that they are specified on the command line.                                                                                                   |

If no command is specified on the command line, `--create` is used by default.



This option is not available in the IDE.

## --delete, -d

|            |                                                                                                                                                                      |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>--delete libraryfile objectfile1 ... objectfileN</code><br><code>-d libraryfile objectfile1 ... objectfileN</code>                                             |
| Parameters | <p><code>libraryfile</code>      The library file that the command operates on. See <i>Rules for specifying a filename or directory as parameters</i>, page 250.</p> |



*objectfile1* ... The object file(s) that the command operates on.  
*objectfileN*

For use with `iarchive`

Description Use this command to remove object files (modules) from an existing library. All object files that are specified on the command line will be removed from the library.



This option is not available in the IDE.

## **--disasm\_data**

Syntax `--disasm_data`

For use with `ielfdump.r178`

Description Use this command to instruct the dumper to dump data sections as if they were code sections.



This option is not available in the IDE.

## **--edit**

Syntax `--edit steering_file`

For use with `isymexport`

Description Use this option to specify a steering file for controlling which symbols are included in the `isymexport` output file, and if desired, also for renaming some of the symbols.

See also *Steering files*, page 469.



This option is not available in the IDE.

## **--export\_locals**

Syntax `--export_locals [=symbol_prefix]`

|              |                      |                                                                                                                                                                                                                                                                         |
|--------------|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Parameters   | <i>symbol_prefix</i> | A custom prefix to the names of exported symbols that replaces the default prefix LOCAL.                                                                                                                                                                                |
| For use with | <i>isymexport</i>    |                                                                                                                                                                                                                                                                         |
| Description  |                      | Use this option to export local symbols from a ROM image file, in addition to absolute symbols. The default name of the exported symbol is <i>LOCAL_filename_symbolname</i> . Use the optional parameter <i>symbol_prefix</i> to replace LOCAL with your custom prefix. |
| Example      |                      | When exported from the ROM image file, the symbol <i> symb </i> in the source file <i>myFile.c</i> becomes <i>LOCAL_myFile_c_symb</i> .                                                                                                                                 |



This option is not available in the IDE.

## **--extract, -x**

|              |                                                                                                                   |                                                                                                                                                                                                                                                 |
|--------------|-------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <i>--extract libraryfile [objectfile1 ... objectfileN]</i><br><i>-x libraryfile [objectfile1 ... objectfileN]</i> |                                                                                                                                                                                                                                                 |
| Parameters   | <i>libraryfile</i>                                                                                                | The library file that the command operates on. See <i>Rules for specifying a filename or directory as parameters</i> , page 250.                                                                                                                |
|              | <i>objectfile1 ... objectfileN</i>                                                                                | The object file(s) that the command operates on.                                                                                                                                                                                                |
| For use with | <i>iarchive</i>                                                                                                   |                                                                                                                                                                                                                                                 |
| Description  |                                                                                                                   | Use this command to extract object files (modules) from an existing library. If a list of object files is specified, only these files are extracted. If a list of object files is not specified, all object files in the library are extracted. |



This option is not available in the IDE.

## **-f**

|            |                    |                                                                                   |
|------------|--------------------|-----------------------------------------------------------------------------------|
| Syntax     | <i>-f filename</i> |                                                                                   |
| Parameters |                    | See <i>Rules for specifying a filename or directory as parameters</i> , page 250. |

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| For use with | <code>iarchive</code> , <code>ielfdump178</code> , <code>iobjmanip</code> , and <code>isymexport</code> .                                                                                                                                                                                                                                                                                                                                                                                                                |
| Description  | <p>Use this option to make the tool read command line options from the named file, with the default filename extension <code>.xcl</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you can 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> |



This option is not available in the IDE.


## --f

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--f filename</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Parameters   | See <i>Rules for specifying a filename or directory as parameters</i> , page 250.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| For use with | <code>iarchive</code> , <code>ielfdump178</code> , <code>iobjmanip</code> , and <code>isymexport</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description  | <p>Use this option to make the tool read command line options from the named file, with the default filename extension <code>.xcl</code>.</p> <p>In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.</p> <p>Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.</p> <p>If you also specify <code>--dependencies</code> on the command line for the tool, extended command line files specified using <code>--f</code> will generate a dependency, but those specified using <code>-f</code> will not generate a dependency.</p> |
| See also     | <code>-f</code> , page 482.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |



This option is not available in the IDE.

## --fill

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--fill [v;]pattern;range[;range...]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Parameters   | <p><i>v</i> Generates virtual fill for the fill command. Virtual fill is filler bytes that are included in checksumming, but that are not included in the output file. The primary use for this is certain types of hardware where bytes that are not specified by the image have a known value—typically, 0xFF or 0x0.</p> <p><i>pattern</i> A hexadecimal string with the 0x prefix, for example, 0xEF, interpreted as a sequence of bytes, where each pair of digits corresponds to one byte, for example 0x123456, for the sequence of bytes 0x12, 0x34, and 0x56. This sequence is repeated over the fill area. If the length of the fill pattern is greater than 1 byte, it is repeated as if it started at address 0.</p> <p><i>range</i> Specifies the address range for the fill. Hexadecimal and decimal notation is allowed, for example, 0x8002-0x8FFF. Note that each address must be 4-byte aligned.</p> <p>Symbols that are present in the ELF file can be used in the range description, for example, __checksum_begin-__checksum_end.</p> |
| For use with | <code>ielftool</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Description  | <p>Use this option to fill all gaps in one or more ranges with a pattern, which can be either an expression or a hexadecimal string. The contents will be calculated as if the fill pattern was repeatedly filled from the start address until the end address is passed, and then the real contents will overwrite that pattern.</p> <p>If the <code>--fill</code> option is used more than once on the command line, the fill ranges cannot overlap each other.</p> <p> To set related options, choose:</p> <p><b>Project&gt;Options&gt;Linker&gt;Checksum</b></p>                                                                                                                                                                                                                                                                                                                                                                                                    |

## --front\_headers

|              |                              |
|--------------|------------------------------|
| Syntax       | <code>--front_headers</code> |
| For use with | <code>ielftool</code>        |

**Description** Use this option to output ELF program and section headers in the beginning of the file, instead of at the end.



This option is not available in the IDE.

## **--generate\_vfe\_header**

**Syntax** `--generate_vfe_header`

**For use with** `isymexport`

**Description** Use this option to declare that the image does not contain any virtual function calls to potentially discarded functions.

When the linker performs virtual function elimination, it discards virtual functions that appear not to be needed. For the optimization to be applied correctly, there must be no virtual function calls in the image that affect the functions that are discarded.

**See also** *Virtual function elimination*, page 119.



To set this options, use:

**Project>Options>Linker>Extra Options**

## **--ihex**

**Syntax** `--ihex`

**For use with** `ielftool`

**Description** Sets the format of the output file to 32-bit linear Intel Extended hex, a hexadecimal text format defined by Intel.



To set related options, choose:

**Project>Options>Linker>Output converter**

## **--no\_bom**

**Syntax** `--no_bom`

**For use with** `iarchive, ielfdump178, iobjmanip, and isymexport`

**Description** Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

**See also** `--text_out`, page 498 and *Text encodings*, page 244



This option is not available in the IDE.

## **--no\_header**

**Syntax** `--no_header`

**For use with** `ielfdump.r178`

**Description** By default, a standard list header is added before the actual file content. Use this option to suppress output of the list header.



This option is not available in the IDE.

## **--no\_rel\_section**

**Syntax** `--no_rel_section`

**For use with** `ielfdump.r178`

**Description** By default, whenever the content of a section of a relocatable file is generated as output, the associated section, if any, is also included in the output. Use this option to suppress output of the relocation section.



This option is not available in the IDE.

## **--no\_strtab**

**Syntax** `--no_strtab`

**For use with** `ielfdump.r178`

**Description** Use this option to suppress dumping of string table sections (sections of type `SHT_STRTAB`).



This option is not available in the IDE.

## --no\_utf8\_in

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--no_utf8_in</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| For use with | <code>ielfdump.r178</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Description  | <p>The dumper can normally determine whether ELF files produced by IAR tools use the UTF-8 text encoding or not, and produce the correct output. For ELF files produced by non-IAR tools, the dumper will assume UTF-8 encoding unless this option is used, in which case the encoding is assumed to be according to the current system default locale.</p> <p><b>Note:</b> This only makes a difference if any characters beyond 7-bit ASCII are used in paths, symbols, etc.</p> |
| See also     | <i>Text encodings</i> , page 244                                                                                                                                                                                                                                                                                                                                                                                                                                                   |



This option is not available in the IDE.

## --offset

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--offset [-]offset</code>                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Parameters   | <p><i>offset</i>                      The offset will be added (or subtracted if - is specified) to all addresses in the generated output file.</p>                                                                                                                                                                                                                                                                                                                        |
| For use with | <code>ielftool</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Description  | <p>Use this option to add or subtract an offset to the address of each output record in the generated output file. The option only works on Motorola S-records, Intel Hex, TI-Txt, and Simple-Code. The option has no effect when generating an ELF file or when binary files (<code>--bin</code> contain no address information) are generated. No content, including the entry point, will be changed by using this option, only the addresses in the output format.</p> |
| Example      | <p><code>--offset 0x30000</code></p> <p>This will add an offset of <code>0x30000</code> to all addresses. As a result, content that was linked at address <code>0x4000</code> will be placed at <code>0x34000</code>.</p>                                                                                                                                                                                                                                                  |



This option is not available in the IDE.

## --output, -o

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>-o {filename directory}</code><br><code>--output {filename directory}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Parameters   | See <i>Rules for specifying a filename or directory as parameters</i> , page 250.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| For use with | <code>iarchive</code> and <code>ielfdumppr178</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Description  | <p><code>iarchive</code></p> <p>By default, <code>iarchive</code> assumes that the first argument after the <code>iarchive</code> command is the name of the destination library. Use this option to explicitly specify a different filename for the library.</p> <p><code>ielfdumppr178</code></p> <p>By default, output from the dumper is directed to the console. Use this option to direct the output to a file instead. The default name of the output file is the name of the input file with an added <code>id</code> filename extension</p> <p>You can also specify the output file by specifying a file or directory following the name of the input file.</p> |



This option is not available in the IDE.

## --parity

|            |                                                                                                      |                                                                                                                                    |
|------------|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>--parity{symbol[+offset]   address} : size, algo: flashbase[: flags]; range[: range...]</code> |                                                                                                                                    |
| Parameters | <i>symbol</i>                                                                                        | The name of the symbol where the parity bytes should be stored. Note that it must exist in the symbol table in the input ELF file. |
|            | <i>offset</i>                                                                                        | An offset to the symbol. By default, 0.                                                                                            |
|            | <i>address</i>                                                                                       | The absolute address where the parity bytes should be stored.                                                                      |



|  |                  |                                                                                                                                                                                                                                                                          |
|--|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <i>size</i>      | The maximum number of bytes that the parity generation can use. An error will be issued if this value is exceeded. Note that the size must fit in the specified symbol in the ELF file.                                                                                  |
|  | <i>algo</i>      | Choose between:<br><br>odd, uses odd parity.<br>even, uses even parity.                                                                                                                                                                                                  |
|  | <i>flashbase</i> | The start address of the flash memory. Parity bits will not be generated for the addresses between <i>flashbase</i> and the start address of the range. If <i>flashbase</i> and the start address of the range coincide, parity bits will be generated for all addresses |
|  | <i>flags</i>     | Choose between:<br><br>r, reverses the byte order within each word.<br>L, processes 4 bytes at a time.<br>W, processes 2 bytes at a time.<br>B, processes 1 byte at a time.                                                                                              |
|  | <i>range</i>     | The address range over which the parity bytes should be generated. Hexadecimal and decimal notation are allowed, for example, 0x8002-0x8FFF.                                                                                                                             |

For use with `ielftool`

Description Use this option to generate parity bytes over specified ranges. The range is traversed left to the right and the parity bits are generated using the odd or even algorithm. The parity bits are finally stored in the specified symbol where they can be accessed by your application.



This option is not available in the IDE.

## **--ram\_reserve\_ranges**

Syntax `--ram_reserve_ranges [=symbol_prefix]`

Parameters *symbol\_prefix* The prefix of symbols created by this option.

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| For use with | <code>isymexport</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Description  | <p>Use this option to generate symbols for the areas in RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter <i>symbol_prefix</i>.</p> <p>Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.</p> <p>If <code>--ram_reserve_ranges</code> is used together with <code>--reserve_ranges</code>, the RAM areas will get their prefix from the <code>--ram_reserve_ranges</code> option and the non-RAM areas will get their prefix from the <code>--reserve_ranges</code> option.</p> |
| See also     | <code>--reserve_ranges</code> , page 493.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |



This option is not available in the IDE.

## **--range**

|              |                                                                                                                                                                                    |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--range start-end</code>                                                                                                                                                     |
| Parameters   | <p><i>start-end</i>                      Disassemble code where the start address is greater than or equal to <i>start</i>, and where the end address is less than <i>end</i>.</p> |
| For use with | <code>ielfdump178</code>                                                                                                                                                           |
| Description  | Use this option to specify a range for which code from an executable will be dumped.                                                                                               |



This option is not available in the IDE.

## **--raw**

|              |                                                                                                                                                                                             |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--raw</code>                                                                                                                                                                          |
| For use with | <code>ielfdump178</code>                                                                                                                                                                    |
| Description  | By default, many ELF sections will be dumped using a text format specific to a particular kind of section. Use this option to dump each selected ELF section using the generic text format. |

The generic text format dumps each byte in the section in hexadecimal format, and where appropriate, as ASCII text.



This option is not available in the IDE.

## --remove\_file\_path

Syntax `--remove_file_path`

For use with `iobjmanip`

Description Use this option to make `iobjmanip` remove information about the directory structure of the project source tree from the generated object file, which means that the file symbol in the ELF object file is modified.

This option must be used in combination with `--remove_section ".comment"`.



This option is not available in the IDE.

## --remove\_section

Syntax `--remove_section {section|number}`

Parameters

*section* The section—or sections, if there are more than one section with the same name—to be removed.

*number* The number of the section to be removed. Section numbers can be obtained from an object dump created using `ielfdump`r178.

For use with `iobjmanip`

Description Use this option to make `iobjmanip` omit the specified section when generating the output file.



This option is not available in the IDE.

## --rename\_section

|              |                                                                                                              |                                                                                                                                        |
|--------------|--------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--rename_section {<i>oldname</i> <i>oldnumber</i>}=<i>newname</i></code>                               |                                                                                                                                        |
| Parameters   | <i>oldname</i>                                                                                               | The section—or sections, if there are more than one section with the same name—to be renamed.                                          |
|              | <i>oldnumber</i>                                                                                             | The number of the section to be renamed. Section numbers can be obtained from an object dump created using <code>ie1fdumpr178</code> . |
|              | <i>newname</i>                                                                                               | The new name of the section.                                                                                                           |
| For use with | <code>iobjmanip</code>                                                                                       |                                                                                                                                        |
| Description  | Use this option to make <code>iobjmanip</code> rename the specified section when generating the output file. |                                                                                                                                        |



This option is not available in the IDE.

## --rename\_symbol

|              |                                                                                                             |                             |
|--------------|-------------------------------------------------------------------------------------------------------------|-----------------------------|
| Syntax       | <code>--rename_symbol <i>oldname</i> =<i>newname</i></code>                                                 |                             |
| Parameters   | <i>oldname</i>                                                                                              | The symbol to be renamed.   |
|              | <i>newname</i>                                                                                              | The new name of the symbol. |
| For use with | <code>iobjmanip</code>                                                                                      |                             |
| Description  | Use this option to make <code>iobjmanip</code> rename the specified symbol when generating the output file. |                             |



This option is not available in the IDE.

## --replace, -r

|        |                                                                                                                                                                     |  |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax | <code>--replace <i>libraryfile</i> <i>objectfile1</i> ... <i>objectfileN</i></code><br><code>-r <i>libraryfile</i> <i>objectfile1</i> ... <i>objectfileN</i></code> |  |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|

|              |                                                                                                                                                                                                                                                                         |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Parameters   | <p><i>libraryfile</i>      The library file that the command operates on. See <i>Rules for specifying a filename or directory as parameters</i>, page 250.</p> <p><i>objectfile1</i> ...    The object file(s) that the command operates on.<br/><i>objectfileN</i></p> |
| For use with | iarchive                                                                                                                                                                                                                                                                |
| Description  | Use this command to replace or add object files (modules) to an existing library. The object files specified on the command line either replace existing object files in the library—if they have the same name—or are appended to the library.                         |



This option is not available in the IDE.

## --reserve\_ranges

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--reserve_ranges[=<i>symbol_prefix</i>]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Parameters   | <p><i>symbol_prefix</i>      The prefix of symbols created by this option.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| For use with | isymexport                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Description  | <p>Use this option to generate symbols for the areas in ROM and RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter <i>symbol_prefix</i>.</p> <p>Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.</p> <p>If <code>--reserve_ranges</code> is used together with <code>--ram_reserve_ranges</code>, the RAM areas will get their prefix from the <code>--ram_reserve_ranges</code> option and the non-RAM areas will get their prefix from the <code>--reserve_ranges</code> option.</p> |
| See also     | <code>--ram_reserve_ranges</code> , page 489.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |



This option is not available in the IDE.

## --section, -s

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--section <i>section_number</i> <i>section_name</i>[,...]</code><br><code>--s <i>section_number</i> <i>section_name</i>[,...]</code>                                                                                                                                                                                                                                                                                                                                                                                                  |
| Parameters   | <i>section_number</i> The number of the section to be dumped.<br><i>section_name</i> The name of the section to be dumped.                                                                                                                                                                                                                                                                                                                                                                                                                  |
| For use with | <code>ielfdump.r178</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Description  | <p>Use this option to dump the contents of a section with the specified number, or any section with the specified name. If a relocation section is associated with a selected section, its contents are output as well.</p> <p>If you use this option, the general properties of the input file will not be included in the output.</p> <p>You can specify multiple section numbers or names by separating them with commas, or by using this option more than once.</p> <p>By default, no section contents are included in the output.</p> |
| Example      | <pre>-s 3,17                               /* Sections #3 and #17 -s .debug_frame,42                   /* Any sections named .debug_frame and                                       also section #42 */</pre>                                                                                                                                                                                                                                                                                                                               |



This option is not available in the IDE.

## --segment, -g

|              |                                                                                                                                    |
|--------------|------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--segment <i>segment_number</i>[,...]</code><br><code>-g <i>segment_number</i>[,...]</code>                                  |
| Parameters   | <i>segment_number</i> The number of a segment whose contents will be included in the output.                                       |
| For use with | <code>ielfdump.r178</code>                                                                                                         |
| Description  | Use this option to select specific segments—parts of an executable image indicated by program headers—for inclusion in the output. |



This option is not available in the IDE.

## --self\_reloc

Syntax `--self_reloc`

For use with `ielftool`

Description This option is intentionally not documented as it is not intended for general use.



This option is not available in the IDE.

## --show\_entry\_as

Syntax `--show_entry_as name`

Parameters *name* The name to give to the program entry point in the output file.

For use with `isymexport`

Description Use this option to export the entry point of the application given as input under the name *name*.



This option is not available in the IDE.

## --silent

Syntax `--silent`

For use with `ielftool`

Description Causes the tool to operate without sending any messages to the standard output stream. By default, the tool sends various messages via the standard output stream. You can use this option to prevent this. The tool sends error and warning messages to the error output stream, so they are displayed regardless of this setting.



This option is not available in the IDE.

## --simple

|              |                                                                                                       |
|--------------|-------------------------------------------------------------------------------------------------------|
| Syntax       | --simple                                                                                              |
| For use with | ielftool                                                                                              |
| Description  | Sets the format of the output file to Simple-code, a binary format that includes address information. |



To set related options, choose:

**Project>Options>Output converter**

## --simple-ne

|              |                                                                                      |
|--------------|--------------------------------------------------------------------------------------|
| Syntax       | --simple-ne                                                                          |
| For use with | ielftool                                                                             |
| Description  | Sets the format of the output file to Simple code, but no entry record is generated. |



To set related options, choose:

**Project>Options>Output converter**

## --source


|              |                                                                                                                                                                                                                                                                                                                     |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | --source                                                                                                                                                                                                                                                                                                            |
| For use with | iefldumpr178                                                                                                                                                                                                                                                                                                        |
| Description  | Use this option to make <code>ielftool</code> include source for each statement before the code for that statement, when dumping code from an executable file. To make this work, the executable image must be built with debug information, and the source code must still be accessible in its original location. |




This option is not available in the IDE.




**--srec**

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--srec</code>                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| For use with | <code>ielftool</code>                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Description  | <p>Sets the format of the output file to Motorola S-records, a hexadecimal text format defined by Motorola.</p> <p><b>Note:</b> You can use the <code>ielftool</code> options <code>--srec-len</code> and <code>--srec-s3only</code> to modify the exact format used.</p> <p> To set related options, choose:<br/><b>Project&gt;Options&gt;Output converter</b></p> |


**--srec-len**

|              |                                                                                                                                                                                                                                                                                                                                        |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--srec-len=length</code>                                                                                                                                                                                                                                                                                                         |
| Parameters   | <p><i>length</i>                      The number of data bytes in each S-record.</p>                                                                                                                                                                                                                                                   |
| For use with | <code>ielftool</code>                                                                                                                                                                                                                                                                                                                  |
| Description  | <p>Sets the maximum number of data bytes in an S-record. This option can only be used together with the <code>--srec</code> option. By default, the number of data bytes in an S-record is 16.</p> <p> This option is not available in the IDE.</p> |


**--srec-s3only**

|              |                                                                                                                                                                                                                                                                                                                          |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--srec-s3only</code>                                                                                                                                                                                                                                                                                               |
| For use with | <code>ielftool</code>                                                                                                                                                                                                                                                                                                    |
| Description  | <p>Restricts the S-record output to contain only a subset of records, that is S0, S3 and S7 records. This option can be used in combination with the <code>--srec</code> option.</p> <p> This option is not available in the IDE.</p> |

## --strip

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--strip</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| For use with | <code>iobjmanip</code> and <code>ielftool</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Description  | <p>Use this option to remove all sections containing debug information before the output file is written.</p> <p><b>Note:</b> <code>ielftool</code> needs an unstripped input ELF image. If you use the <code>--strip</code> option in the linker, remove it and use the <code>--strip</code> option in <code>ielftool</code> instead.</p> <p> To set related options, choose:</p> <p><b>Project&gt;Options&gt;Linker&gt;Output&gt;Include debug information in output</b></p> |

## --symbols

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--symbols libraryfile</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Parameters   | <p><i>libraryfile</i>      The library file that the command operates on. See <i>Rules for specifying a filename or directory as parameters</i>, page 250.</p>                                                                                                                                                                                                                                                                                                                                         |
| For use with | <code>iarchive</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Description  | <p>Use this command to list all external symbols that are defined by any object file (module) in the specified library, together with the name of the object file (module) that defines it.</p> <p>In silent mode (<code>--silent</code>), this command performs symbol table-related syntax checks on the library file and displays only errors and warnings.</p> <p> This option is not available in the IDE.</p> |

## --text\_out


|            |                                                                                                                                                             |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>--text_out {utf8 utf16le utf16be locale}</code>                                                                                                       |
| Parameters | <p><code>utf8</code>                      Uses the UTF-8 encoding</p> <p><code>utf16le</code>                    Uses the UTF-16 little-endian encoding</p> |

|              |                                                                                                                                                                                                                                                                                                                                                                                   |                                     |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|
|              | <code>utf16be</code>                                                                                                                                                                                                                                                                                                                                                              | Uses the UTF-16 big-endian encoding |
|              | <code>locale</code>                                                                                                                                                                                                                                                                                                                                                               | Uses the system locale encoding     |
| For use with | <code>iarchive</code> , <code>ielfdumpr178</code> , <code>iobjmanip</code> , and <code>isymexport</code>                                                                                                                                                                                                                                                                          |                                     |
| Description  | Use this option to specify the encoding to be used when generating a text output file.<br><br>The default for the list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM).<br><br>If you want text output in UTF-8 encoding without BOM, you can use the option <code>--no_bom</code> as well. |                                     |
| See also     | <code>--no_bom</code> , page 485 and <i>Text encodings</i> , page 244                                                                                                                                                                                                                                                                                                             |                                     |



This option is not available in the IDE.

## --titxt

|              |                                                                                                                                                                |  |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax       | <code>--titxt</code>                                                                                                                                           |  |
| For use with | <code>ielftool</code>                                                                                                                                          |  |
| Description  | Sets the format of the output file to Texas Instruments TI-TXT, a hexadecimal text format defined by Texas Instruments.<br><br>To set related options, choose: |  |
|              |  <b>Project&gt;Options&gt;Output converter</b>                              |  |

## --toc, -t

|              |                                                                                          |                                                                                                                                  |
|--------------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>--toc libraryfile</code><br><code>-t libraryfile</code>                            |                                                                                                                                  |
| Parameters   | <code>libraryfile</code>                                                                 | The library file that the command operates on. See <i>Rules for specifying a filename or directory as parameters</i> , page 250. |
| For use with | <code>iarchive</code>                                                                    |                                                                                                                                  |
| Description  | Use this command to list the names of all object files (modules) in a specified library. |                                                                                                                                  |

In silent mode (`--silent`), this command performs basic syntax checks on the library file, and displays only errors and warnings.



This option is not available in the IDE.

## **--use\_full\_std\_template\_names**

Syntax `--use_full_std_template_names`

For use with `ielfdump``rl78`

Description Normally, the names of some standard C++ templates are used in the output in an abbreviated form in the unmangled names of symbols, for example, "`std::string`" instead of "`std::basic_string<char, std::char_traits<char>, std::allocator<char>>`". Use this option to make `ielfdump` use the unabbreviated form.



This option is not available in the IDE.

## **--utf8\_text\_in**

Syntax `--utf8_text_in`

For use with `iarchive`, `ielfdump``rl78`, `iobjmanip`, and `isymexport`

Description Use this option to specify that the tool shall use the UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).

**Note:** This option does not apply to source files.

See also *Text encodings*, page 244



This option is not available in the IDE.

## **--verbose, -V**

Syntax `--verbose`  
`-V` (`iarchive` only)

For use with `iarchive` and `ielftool`.

**Description** Use this option to make the tool report which operations it performs, in addition to giving diagnostic messages.



This option is not available in the IDE because this setting is always enabled.

## --version

**Syntax** `--version`

**For use with** `iarchive, ielfdump178, ielftool, iobjmanip, isymexport`

**Description** Use this option to make the tool send version information to the console and then exit.



This option is not available in the IDE.

## --vtoc

**Syntax** `--vtoc libraryfile`

**Parameters**

`libraryfile` The library file that the command operates on. See *Rules for specifying a filename or directory as parameters*, page 250.

**For use with** `iarchive`

**Description** Use this command to list the names, sizes, and modification times of all object files (modules) in a specified library.

In silent mode (`--silent`), this command performs basic syntax checks on the library file, and displays only errors and warnings.



This option is not available in the IDE.



# Implementation-defined behavior for Standard C++

- Descriptions of implementation-defined behavior for C++
- Implementation quantities

If you are using C instead of C++, see *Implementation-defined behavior for Standard C*, page 523 or *Implementation-defined behavior for C89*, page 543, respectively.

---

## Descriptions of implementation-defined behavior for C++

This section follows the same order as the C++ 14 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++ 14. This means that parts of a standard library can be excluded from the implementation.

### I GENERAL

#### Diagnostics (I.3.6)

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.

#### Required libraries for freestanding implementation (I.4)

See *C++ header files*, page 397 and *Not supported C/C++ functionality*, page 401, respectively, for information about which Standard C++ system headers that the IAR C/C++ Compiler does not support.

### Bits in a byte (1.7)

A byte contains 8 bits.

### Interactive devices (1.9)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

### Number of threads in a program under a freestanding implementation (1.10)

By default, the IAR Systems runtime environment does not support more than one thread of execution. With an optional third-party RTOS, it might support several threads of execution.

## 2 LEXICAL CONVENTIONS

### Mapping physical source file characters to the basic source character set (2.2)

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, it can be UTF-8, UTF-16, or the system locale. See *Text encodings*, page 244.

### Physical source file characters (2.2)

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, it can be UTF-8, UTF-16, or the system locale. See *Text encodings*, page 244.

### Converting characters from a source character set to the execution character set (2.2)

The source character set is the set of legal characters that can appear in source files. It is dependent on the chosen encoding for the source file. See *Text encodings*, page 244. By default, the source character set is Raw.

The execution character set is the set of legal characters that can appear in the execution environment. These are the execution character sets for character constants and string literals, and their encoding types:

| Execution character set | Encoding type |
|-------------------------|---------------|
| L                       | UTF-32        |
| u                       | UTF-16        |
| U                       | UTF-32        |

Table 50: Execution character sets and their encodings



| Execution character set | Encoding type            |
|-------------------------|--------------------------|
| u8                      | UTF-8                    |
| none                    | The source character set |

Table 50: Execution character sets and their encodings (Continued)

The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 155.

### Required availability of the source of translation units to locate template definitions (2.2)

When locating the template definition related to template instantiations, the source of the translation units that define the template is not required.

### The execution character set and execution wide-character set (2.3)

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 source file character set. The source file character set is determined by the chosen encoding for the source file. See *Text encodings*, page 244.

The wide character set consists of all the code points defined by ISO/IEC 10646.

### Mapping header names to headers or external source files (2.9)

The header name is interpreted and mapped into an external source file in the most intuitive way. In both forms of the `#include` preprocessing directive, the character sequences that specify header names are interpreted exactly in the same way as for other source constructs. They are then mapped to external header source file names.

### The value of multi-character literals (2.14.3)

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 is issued if the value cannot be represented in an integer constant.

### The value of wide-character literals with single c-char that are not in the execution wide-character set (2.14.3)

All possible c-chars have a representation in the execution wide-character set.

### **The value of wide-character literal containing multiple characters (2.14.3)**

A diagnostic message is issued, and all but the first c-char is ignored.

### **The semantics of non-standard escape sequences (2.14.3)**

No non-standard escape sequences are supported.

### **The value of character literal outside range of corresponding type (2.14.3)**

The value is truncated to fit the type.

### **The encoding of universal character name not in execution character set (2.14.3)**

A diagnostic message is issued.

### **The choice of larger or smaller value of floating-point literal (2.14.4)**

For a floating-point literal whose scaled value cannot be represented as a floating-point value, the nearest even floating point-value is chosen.

### **The distinctness of string literals (2.14.5)**

All string literals are distinct except when the linker option `--merge_duplicate_sections` is used.

### **Concatenation of various types of string literals (2.14.5)**

Differently prefixed string literal tokens cannot be concatenated, except for those specified by the ISO C++ standard.

## **3 BASIC CONCEPTS**

### **Defining main in a freestanding environment (3.6.1)**

The `main` function must be defined.

### **Startup and termination in a freestanding environment (3.6.1)**

See *Application execution—an overview*, page 56 and *System startup and termination*, page 141, for descriptions of the startup and termination of applications.

### Parameters to main (3.6.1)

The only two permitted definitions for `main` are:

```
int main()
int main(int, char **)
```

### Linkage of main (3.6.1)

The `main` function has external linkage.

### Dynamic initialization of static objects before main (3.6.2)

Static objects are initialized before the first statement of `main`, except when the linker option `--manual_dynamic_initialization` is used.

### Dynamic initialization of threaded local objects before entry (3.6.2)

By default, the IAR systems runtime environment does not support more than one thread of execution. With an optional third-party RTOS, it might support several threads of execution.

Thread-local objects are treated as static objects except when the linker option `--threaded_lib` is used. Then they are initialized by the RTOS.

### Use of an invalid pointer (3.7.4.2)

Any other use of an invalid pointer than indirection through it and passing it to a deallocation function works as for a valid pointer.

### Relaxed or strict pointer safety for the implementation (3.7.4.3)

The IAR Systems implementation of Standard C++ has relaxed pointer safety.

### The value of trivially copyable types (3.9)

All bits in basic types are part of the value representation. Padding between basic types is copied verbatim.

### Representation and signage of char (3.9.1)

A plain `char` is treated as an `unsigned char`. See `--char_is_signed`, page 257 and `--char_is_unsigned`, page 257.

### Extended signed integer types (3.9.1)

No extended signed integer types exist in the implementation.

### **Value representation of floating-point types (3.9.1)**

See *Basic data types—floating-point types*, page 326.

### **Value representation of pointer types (3.9.2)**

See *Pointer types*, page 328.

### **Alignment (3.11)**

See *Alignment*, page 321.

### **Alignment additional values (3.11)**

See *Alignment*, page 321.

### **alignof expression additional values (3.11)**

See *Alignment*, page 321.

## **4 STANDARD CONVERSIONS**

### **lvalue-to-rvalue conversion for objects that contain an invalid pointer (4.1)**

The conversion is made as if the pointer was valid.

### **The value of the result of unsigned to signed conversion (4.7)**

When an integer value is converted to a value of signed integer type, but cannot be represented by the destination type, the value is truncated to the number of bits of the destination type and then reinterpreted as a value of the destination type.

### **The result of inexact floating-point conversion (4.8)**

When a floating-point value is converted to a value of a different floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

### **The value of the result of an inexact integer to floating-point conversion (4.9)**

When an integer value is converted to a value of a floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

**The rank of extended signed integer types (4.13)**

The implementation has no extended signed integer types.

**5 EXPRESSIONS****Passing argument of class type through ellipsis (5.2.2)**

The result is a diagnostic and is then treated as a trivially copyable object.

**The derived type for typeid (5.2.8)**

The type of a `typeid` expression is an expression with dynamic type `std::type_info`.

**Conversion from a pointer to an integer (5.2.10)**

See *Casting*, page 329.

**Conversion from an integer to a pointer (5.2.10)**

See *Casting*, page 329.

**Converting a function pointer to an object pointer and vice versa (5.2.10)**

See *Casting*, page 329.

**sizeof applied to fundamental types other than char, signed char, and unsigned char (5.3.3)**

See *Basic data types—integer types*, page 322, *Basic data types—floating-point types*, page 326, and *Pointer types*, page 328.

**Support for over-aligned types (5.3.4)**

Over-aligned types are supported in `new` expressions.

**The type of ptrdiff\_t (5.7)**

See *ptrdiff\_t*, page 330.

**The result of right shift of negative value (5.8)**

In a bitwise right shift operation of the form `E1 >> E2`, if `E1` is of signed type and has a negative value, the value of the result is the integral part of the quotient  $E1 / (2^{**E2})$ , except when `E1` is `-1`.

## 7 DECLARATIONS

### The meaning of the attribute declaration (7)

There are no other attributes supported than what is specified in the C++ standard. See *Extended keywords*, page 335, for supported attributes and ways to use them with objects.

### Access to an object that has volatile-qualified type (7.1.6.1)

See *Declaring objects volatile*, page 332.

### The underlying type for enumeration (7.2)

See *The enum type*, page 323.

### The meaning of the asm declaration (7.4)

An `asm` declaration enables the direct use of assembler instructions.

### The semantics of linkage specifiers (7.5)

Only the string-literals "C" and "C++" can be used in a linkage specifier.

### Linkage of objects to other languages than C (7.5)

The IAR Systems implementation of Standard C++ does not support linkage to other languages than C.

### The behavior of attribute-scoped tokens (7.6.1)

The use of an attribute-scoped token is not supported.

### The behavior of non-standard attributes (7.6.1)

There are no other attributes supported other than what is specified in the C++ standard. See *Extended keywords*, page 335, for a list supported attributes and ways to use them with objects.

## 8 DECLARATORS

### The string resulting from `__func__` (8.4.1)

The value of `__func__` is the C++ function name.

## 9 CLASSES

### Allocation of bitfields within a class object (9.6)

See *Bitfields*, page 324.

## 14 TEMPLATES

### The semantics of linkage specification on templates (14)

Only the string-literals "C" and "C++" can be used in a linkage specifier.

## 15 EXCEPTION HANDLING

### Stack unwinding before calling `std::terminate()` (15.3, 15.5.1)

When no suitable catch handler is found, the stack is not unwound before calling `std::terminate()`.

### Stack unwinding before calling `std::terminate()` when a `noexcept` specification is violated (15.5.1)

When a `noexcept` specification is violated, the stack is not unwound before calling `std::terminate()`.

### Bad throw in `std::unexpected` (15.5.2)

If `std::unexpected` throws an exception that is not allowed by the exception specification for the function that caused the original exception specification violation, and that exception specification includes `std::bad_exception`, then the thrown exception is replaced by a `std::bad_exception` and the search for another handler continues.

## 16 PREPROCESSING DIRECTIVES

### The numeric values of character literals in `#if` directives (16.1)

Numeric values of character literals in the `#if` and `#elif` preprocessing directives match the values that they have in other expressions.

### Negative value of character literal in preprocessor (16.1)

A plain `char` is treated as an `unsigned char`. See *--char\_is\_signed*, page 257 and *--char\_is\_unsigned*, page 257. If a `char` is treated as a signed character, then character literals in `#if` and `#elif` preprocessing directives can be negative.

### **Search locations for < > header (16.2)**

See *Include file search procedure*, page 241.

### **The search procedure for included source file (16.2)**

See *Include file search procedure*, page 241.

### **Search locations for "" header (16.2)**

See *Include file search procedure*, page 241.

### **The sequence of places searched for a header (16.2)**

See *Include file search procedure*, page 241.

### **Nesting limit for #include directives (16.2)**

The amount of available memory sets the limit.

### **#pragma (16.6)**

See *Recognized pragma directives (6.10.6)*, page 531.

### **The definition and meaning of \_\_STDC\_\_ (16.8)**

\_\_STDC\_\_ is predefined to 1.

### **The text of \_\_DATE\_\_ when date of translation is not available (16.8)**

The date of the translation is always available.

### **The text of \_\_TIME\_\_ when time of translation is not available (16.8)**

The time of the translation is always available.

### **The definition and meaning of \_\_STDC\_VERSION\_\_ (16.8)**

\_\_STDC\_VERSION\_\_ is predefined to 201112L.

## **17 LIBRARY INTRODUCTION**

### **Headers for a freestanding implementation (17.6.1.3)**

See *DLIB runtime environment—implementation details*, page 395.



**Linkage of names from Standard C library (17.6.2.3)**

Declarations from the C library have "C" linkage.

**Functions in Standard C++ library that can be recursively reentered (17.6.5.8)**

Functions can be recursively reentered, unless specified otherwise by the ISO C++ standard.

**Exceptions thrown by standard library functions that do not have an exception specification (17.6.5.12)**

These functions do not throw any additional exceptions.

**error\_category for errors originating outside of the operating system (17.6.5.14)**

There is no additional error category.

**18 LANGUAGE SUPPORT LIBRARY****Definition of NULL (18.2)**

NULL is predefined as 0.

**The type of ptrdiff\_t (18.2)**

See *ptrdiff\_t*, page 330.

**The type of size\_t (18.2)**

See *size\_t*, page 329.

**Exit status (18.5)**

Control is returned to the `__exit` library function. See *\_\_exit*, page 147.

**The return value of `bad_alloc::what` (18.6.2.1)**

The return value is a pointer to "bad allocation".

**The return value of `bad_array_new_length::what` (18.6.2.2)**

The return value is a pointer to "bad allocation".

### **The return value of `type_info::name()` (18.7.1)**

The return value is a pointer to a C string containing the name of the type.

### **The return value of `bad_cast::what` (18.7.2)**

The return value is a pointer to "bad cast".

### **The return value of `bad_typeid::what` (18.7.3)**

The return value is a pointer to "bad typeid".

### **The result of `exception::what` (18.8.1)**

The return value is a pointer to "unknown".

### **The return value of `bad_exception::what` (18.8.2)**

The return value is a pointer to "bad exception".

### **The use of non-POF functions as signal handlers (18.10)**

Non-Plain Old Functions (POF) can be used as signal handlers if no uncaught exceptions are thrown in the handler, and if the execution of the signal handler does not trigger undefined behavior.

## **20 GENERAL UTILITIES LIBRARY**

### **`get_pointer_safety` returning `pointer_safety::relaxed` or `pointer_safety::preferred` when the implementation has relaxed pointer safety (20.7.4)**

The function `get_pointer_safety` always returns `std::pointer_safety::relaxed`.

### **Support for over-aligned types (20.7.9.1, 20.7.11)**

Over-aligned types are supported.

### **The exception type when a `shared_ptr` constructor fails (20.8.2.2.1)**

Only `std::bad_alloc` is thrown.

### **The assignability of placeholder objects (20.9.9.1.4)**

Placeholder objects are `CopyAssignable`.

**Support for extended alignment (20.10.7.6)**

Extended alignment is supported.

**Rounding or truncating values to the required precision when converting between `time_t` values and `time_point` objects (20.12.7.1)**

Values are truncated to the required precision when converting between `time_t` values and `time_point` objects.

**21 STRINGS LIBRARY****The type of `streampos` (21.2.3.1)**

The type of `streampos` is `std::fpos<mbstate_t>`.

**The type of `streamoff` (21.2.3.1)**

The type of `streamoff` is `long`.

**Supported multibyte character encoding rules (21.2.3.1)**

See *Locale*, page 155.

**The type of `u16streampos` (21.2.3.2)**

The type of `u16streampos` is `streampos`.

**The return value of `char_traits<char16_t>::eof` (21.2.3.2)**

The return value of `char_traits<char16_t>::eof` is EOF.

**The type of `u32streampos` (21.2.3.3)**

The type of `u32streampos` is `streampos`.

**The return value of `char_traits<char32_t>::eof` (21.2.3.3)**

The return value of `char_traits<char32_t>::eof` is EOF.

**The type of `wstreampos` (21.2.3.4)**

The type of `wstreampos` is `streampos`.

**The return value of `char_traits<wchar_t>::eof` (21.2.3.3)**

The return value of `char_traits<wchar_t>::eof` is EOF.

## 22 LOCALIZATION LIBRARY

### Locale object being global or per-thread (22.3.1)

There is one global locale object for the entire application.

### Locale names (22.3.1.2)

See *Locale*, page 155.

### The effects on the C locale of calling locale::global (22.3.1.5)

Calling this function with an unnamed locale has no effect.

### The value of ctype<char>::table\_size (22.4.1.3)

The value of `ctype<char>::table_size` is 256.

### Additional formats for time\_get::do\_get\_date (22.4.5.1.2)

No additional formats are accepted for `time_get::do_get_date`.

### time\_get::do\_get\_year and two-digit year numbers (22.4.5.1.2)

Two-digit year numbers are accepted by `time_get::do_get_year`. Years from 0 to 68 are parsed as meaning 2000 to 2068, and years from 69 to 99 are parsed as meaning 1969 to 1999.

### Formatted character sequences generated by time\_put::do\_put in the C locale (22.4.5.3.1)

The behavior is the same as that of the library function `strftime`.

### Mapping from name to catalog when calling messages::do\_open (22.4.7.1.2)

No mapping occurs because this function does not open a catalog.

### Mapping to message when calling messages::do\_get (22.4.7.1.2)

No mapping occurs because this function does not open a catalog. `dflt` is returned.

### Mapping to message when calling messages::do\_close (22.4.7.1.2)

The function cannot be called because no catalog can be open.

## 23 CONTAINERS LIBRARY

### The type of `array::iterator` (23.3.2.1)

The type of `array::iterator` is `T *`.

### The type of `array::const_iterator` (23.3.2.1)

The type of `array::const_iterator` is `T const *`.

### The default number of buckets in `unordered_map` (23.5.4.2)

The IAR C/C++ Compiler for RL78 makes a default construction of the `unordered_map` before inserting the elements.

### The default number of buckets in `unordered_multimap` (23.5.5.2)

The IAR C/C++ Compiler for RL78 makes a default construction of the `unordered_multimap` before inserting the elements.

### The default number of buckets in `unordered_set` (23.5.6.2)

The IAR C/C++ Compiler for RL78 makes a default construction of the `unordered_set` before inserting the elements.

### The default number of buckets in `unordered_multiset` (23.5.7.2)

The IAR C/C++ Compiler for RL78 makes a default construction of the `unordered_multiset` before inserting the elements.

## 25 ALGORITHMS LIBRARY

### The underlying source of random numbers for `random_shuffle` (25.3.12)

The underlying source is `rand()`.

## 27 INPUT/OUTPUT LIBRARY

### The behavior of `iostream` classes when `traits::pos_type` is not `streampos` or when `traits::off_type` is not `streamoff` (27.2.2)

No specific behavior has been implemented for this case.

### **The effects of calling `ios_base::sync_with_stdio` after any input or output operation on standard streams (27.5.3.4)**

Previous input/output is not handled in any special way.

### **Argument values to construct `basic_ios::failure` (27.5.5.4)**

When `basic_ios::clear` throws an exception, it throws an exception of type `basic_ios::failure` constructed with the `badbit/failbit/eofbit` set.

### **The `basic_stringbuf` move constructor and the copying of sequence pointers (27.8.2.1)**

The constructor copies the sequence pointers.

### **The effects of calling `basic_streambuf::setbuf` with non-zero arguments (27.8.2.4)**

This function has no effect.

### **The `basic_filebuf` move constructor and the copying of sequence pointers (27.9.1.2)**

The constructor copies the sequence pointers.

### **The effects of calling `basic_filebuf::setbuf` with non-zero arguments (27.9.1.5)**

This will offer the buffer to the C stream by calling `setvbuf()` with the associated file. If anything goes wrong, the stream is reinitialized.

### **The effects of calling `basic_filebuf::sync` when a `get` area exists (27.9.1.5)**

A `get` area cannot exist.

## **28 REGULAR EXPRESSIONS LIBRARY**

### **The type of `regex_constants::error_type` (28.5.3)**

The type is an `enum`. See *The enum type*, page 323.

## 29 ATOMIC OPERATIONS LIBRARY

### The values of various `ATOMIC_..._LOCK_FREE` macros (29.4)

In cases where atomic operations are supported, these macros will have the value 2. See *Atomic operations*, page 401.

## 30 THREAD SUPPORT LIBRARY

### The presence and meaning of `native_handle_type` and `native_handle` (30.2.3)

The `thread` system header is not supported.

## ANNEX D (NORMATIVE): COMPATIBILITY FEATURES

### The type of `ios_base::streamoff` (D.6)

The type of `ios_base::streamoff` is `std::streamoff`.

### The type of `ios_base::streampos` (D.6)

The type of `ios_base::streampos` is `std::streampos`.

---

## Implementation quantities

The IAR Systems implementation of C++ is, like all implementations, limited in the size of the applications it can successfully process.

These limitations apply:

| C++ feature                                                                                                                       | Limitation              |
|-----------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| Nesting levels of compound statements, iteration control structures, and selection control structures.                            | Limited only by memory. |
| Nesting levels of conditional inclusion.                                                                                          | Limited only by memory. |
| Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration. | Limited only by memory. |
| Nesting levels of parenthesized expressions within a full-expression.                                                             | Limited only by memory. |
| Number of characters in an internal identifier or macro name.                                                                     | Limited only by memory. |

*Table 51: C++ implementation quantities*

| <b>C++ feature</b>                                                                     | <b>Limitation</b>                                |
|----------------------------------------------------------------------------------------|--------------------------------------------------|
| Number of characters in an external identifier.                                        | Limited only by memory.                          |
| External identifiers in one translation unit.                                          | Limited only by memory.                          |
| Identifiers with block scope declared in a block.                                      | Limited only by memory.                          |
| Macro identifiers simultaneously defined in one translation unit.                      | Limited only by memory.                          |
| Parameters in one function definition.                                                 | Limited only by memory.                          |
| Arguments in one function call.                                                        | Limited only by memory.                          |
| Parameters in one macro definition.                                                    | Limited only by memory.                          |
| Arguments in one macro invocation.                                                     | Limited only by memory.                          |
| Characters in one logical source line.                                                 | Limited only by memory.                          |
| Characters in a string literal (after concatenation).                                  | Limited only by memory.                          |
| Size of an object.                                                                     | Limited only by memory.                          |
| Nesting levels for <code>#include</code> files.                                        | Limited only by memory.                          |
| Case labels for a switch statement (excluding those for any nested switch statements). | Limited only by memory.                          |
| Data members in a single class.                                                        | Limited only by memory.                          |
| Enumeration constants in a single enumeration.                                         | Limited only by memory.                          |
| Levels of nested class definitions in a single member-specification.                   | Limited only by memory.                          |
| Functions registered by <code>atexit</code> .                                          | Limited by heap memory in the built application. |
| Functions registered by <code>at_quick_exit</code> .                                   | Limited by heap memory in the built application. |
| Direct and indirect base classes.                                                      | Limited only by memory.                          |
| Direct base classes for a single class.                                                | Limited only by memory.                          |
| Members declared in a single class.                                                    | Limited only by memory.                          |
| Final overriding virtual functions in a class, accessible or not.                      | Limited only by memory.                          |
| Direct and indirect virtual bases of a class.                                          | Limited only by memory.                          |
| Static members of a class.                                                             | Limited only by memory.                          |
| Friend declarations in a class.                                                        | Limited only by memory.                          |
| Access control declarations in a class.                                                | Limited only by memory.                          |
| Member initializers in a constructor definition.                                       | Limited only by memory.                          |

*Table 51: C++ implementation quantities (Continued)*



| <b>C++ feature</b>                                                                                              | <b>Limitation</b>                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| Scope qualifiers of one identifier.                                                                             | Limited only by memory.                                                                                                    |
| Nested external specifications.                                                                                 | Limited only by memory.                                                                                                    |
| Recursive <code>constexpr</code> function invocations.                                                          | 1000. This limit can be changed by using the compiler option <code>--max_cost_constexpr_call</code> .                      |
| Full-expressions evaluated within a core constant expression.                                                   | Limited only by memory.                                                                                                    |
| Template arguments in a template declaration.                                                                   | Limited only by memory.                                                                                                    |
| Recursively nested template instantiations, including substitution during template argument deduction (14.8.2). | 64 for a specific template. This limit can be changed by using the compiler option <code>--pending_instantiations</code> . |
| Handlers per try block.                                                                                         | Limited only by memory.                                                                                                    |
| Throw specifications on a single function declaration.                                                          | Limited only by memory.                                                                                                    |
| Number of placeholders (20.9.9.1.4).                                                                            | 20 placeholders from <code>_1</code> to <code>_20</code> .                                                                 |

*Table 51: C++ implementation quantities (Continued)*



# 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 543.

---

## 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, it can be UTF-8, UTF-16, or the system locale. See *Text encodings*, page 244.

### 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, see *System initialization*, page 144.

### 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.

### Multi-threaded environment (5.1.2.4)

By default, the IAR Systems runtime environment does not support more than one thread of execution. With an optional third-party RTOS, it might support several threads of execution.

### 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 (7.22.4.6)**

In the DLIB runtime environment, there are no implementation-defined environment names that are used by the `getenv` function.

### **The system function (7.22.4.8)**

The `system` function is not supported.

## **J.3.3 IDENTIFIERS**

### **Multibyte characters in identifiers (6.4.2)**

Additional multibyte characters may appear in identifiers depending on the chosen encoding for the source file. The supported multibyte characters must be translatable to one Universal Character Name (UCN).

### **Significant characters in identifiers (5.2.4.1, 6.4.2)**

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 source file character set. The source file character set is determined by the chosen encoding for the source file. See *Text encodings*, page 244.

### 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`. See `--char_is_signed`, page 257 and `--char_is_unsigned`, page 257.

### 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. It is dependent on the chosen encoding for the source file. See *Text encodings*, page 244. By default, the source character set is Raw.

The execution character set is the set of legal characters that can appear in the execution environment. These are the execution character set for character constants and string literals and their encoding types:

| Execution character set | Encoding type            |
|-------------------------|--------------------------|
| <code>L</code>          | UTF-32                   |
| <code>u</code>          | UTF-16                   |
| <code>U</code>          | UTF-32                   |
| <code>u8</code>         | UTF-8                    |
| <code>none</code>       | The source character set |

Table 52: Execution character sets and their encodings

The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 155.

### 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)**

See *Source and execution character sets (6.4.4.4, 5.1.1.2)*, page 526.

**Concatenating wide string literals with different encoding types (6.4.5)**

Wide string literals with different encoding types cannot be concatenated.

**Locale used for wide string literals (6.4.5)**

See *Source and execution character sets (6.4.4.4, 5.1.1.2)*, page 526.

**Source characters as executive characters (6.4.5)**

All source characters can be represented as executive characters.

**Encoding of `wchar_t`, `char16_t`, and `char32_t` (6.10.8.2)**

`wchar_t` has the encoding UTF-32, `char16_t` has the encoding UTF-16, and `char32_t` has the encoding UTF-32.

**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 322.

**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.

### **Accuracy of floating-point conversions (5.2.4.2.2)**

The accuracy of floating-point conversions 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 integer 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 329.

### **ptrdiff\_t (6.5.6)**

For information about `ptrdiff_t`, see *ptrdiff\_t*, page 330.

## **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 83.

## **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 324.

### **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 267.

### **Atomic types for bitfields (6.7.2.1)**

Atomic types cannot be used as bitfields.

**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 324.

**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 321.

**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 332.

**J.3.11 PREPROCESSING DIRECTIVES****Locations in #pragma for header names (6.4, 6.4.7)**

These pragma directives take header names as parameters at the specified positions:

```
#pragma include_alias ("header", "header")
#pragma include_alias (<header>, <header>)
```

**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 385.

**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 257.

**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 241.

**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 241.

**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.

**# inserts \ in front of \u (6.10.3.2)**

`#` (stringify argument) inserts a `\` character in front of a Universal Character Name (UCN) in character constants and string literals.

**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.

```
alias_def
alignment
alternate_target_def
baseaddr
building_runtime
can_instantiate
codeseg
```

context\_handler  
cplusplus\_neutral  
cspy\_support  
cstat\_dump  
define\_type\_info  
do\_not\_instantiate  
early\_dynamic\_initialization  
exception\_neutral  
function  
function\_category  
function\_effects  
hdrstop  
important\_typedef  
ident  
implements\_aspect  
init\_routines\_only\_for\_needed\_variables  
initialization\_routine  
inline\_template  
instantiate  
keep\_definition  
library\_default\_requirements  
library\_provides  
library\_requirement\_override  
memory  
module\_name  
no\_pch  
no\_vtable\_use  
once  
pop\_macro

```

preferred_typedef
push_macro
separate_init_routine
set_generate_entries_without_bounds
system_include
uses_aspect
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 121.

### 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 402.

### `feraiseexcept` raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 327.

### Strings passed to the `setlocale` function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 155.

### 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 sets `errno` to a domain error and returns a floating-point NaN when the second argument is zero.

### remainder return value (7.12.10.2)

The `remainder` function sets `errno` to a domain error and 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`.

### remquo return value (7.12.10.3)

The `remquo` function sets `errno` to a domain error and returns a floating-point NaN when the second argument is zero.

### 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 152 and *raise*, page 150, respectively.

### NULL macro (7.19)

The `NULL` macro is defined to 0.

### Terminating newline character (7.21.2)

Stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

**Space characters before a newline character (7.21.2)**

Space characters written to a stream immediately before a newline character are preserved.

**Null characters appended to data written to binary streams (7.21.2)**

No null characters are appended to data written to binary streams.

**File position in append mode (7.21.3)**

The file position is initially placed at the beginning of the file when it is opened in append-mode.

**Truncation of files (7.21.3)**

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 122.

**File buffering (7.21.3)**

An open file can be either block-buffered, line-buffered, or unbuffered.

**A zero-length file (7.21.3)**

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

**Legal file names (7.21.3)**

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 (7.21.3)**

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 (7.21.3)**

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

**remove() (7.21.4.1)**

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 122.

**rename() (7.21.4.2)**

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 122.

**Removal of open temporary files (7.21.4.3)**

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

**Mode changing (7.21.5.4)**

`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 (7.21.6.1, 7.29.2.1)**

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() (7.21.6.1, 7.29.2.1)**

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 (7.21.6.2, 7.29.2.1)**

A `-` (dash) character is always treated as a range symbol.

**%p in scanf (7.21.6.2, 7.29.2.2)**

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 (7.21.9.1, 7.21.9.3, 7.21.9.4)**

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.



**An n-char-sequence after nan (7.22.1.3, 7.29.4.1.1)**

An n-char-sequence after a NaN is read and ignored.

**errno value at underflow (7.22.1.3, 7.29.4.1.1)**

`errno` is set to `ERANGE` if an underflow is encountered.

**Zero-sized heap objects (7.22.3)**

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

**Behavior of abort and exit (7.22.4.1, 7.22.4.5)**

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

**Termination status (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7)**

The termination status will be propagated to `__exit()` as a parameter. `exit()`, `_Exit()`, and `quick_exit` use the input parameter, whereas `abort` uses `EXIT_FAILURE`.

**The system function return value (7.22.4.8)**

The `system` function returns `-1` when its argument is not a null pointer.

**Range and precision of clock\_t and time\_t (7.27)**

The range and precision of `clock_t` is up to your implementation. The range and precision of `time_t` is 19000101 up to 20351231 in tics of a second if the 32-bit `time_t` is used. It is -9999 up to 9999 years in tics of a second if the 64-bit `time_t` is used. See *time.h*, page 403

**The time zone (7.27.1)**

The local time zone and daylight savings time must be defined by the application. For more information, see *time.h*, page 403.

**The era for clock() (7.27.2.1)**

The era for the `clock` function is up to your implementation.

**TIME\_UTC epoch (7.27.2.5)**

The epoch for `TIME_UTC` is up to your implementation.

### **%Z replacement string (7.27.3.5, 7.29.5.1)**

By default, ":" or "" (an empty string) is used as a replacement for %Z. Your application should implement the time zone handling. See `__time32`, `__time64`, page 153.

### **Math functions rounding mode (F.10)**

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, 7.20.2, 7.20.3)**

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 321.

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).

### **Accessing another thread's autos or thread locals (6.2.4)**

The IAR Systems runtime environment does not allow multiple threads. With a third-party RTOS, the access will take place and work as intended as long as the accessed item has not gone out of its scope.

### **The number, order, and encoding of bytes (6.2.6.1)**

See *Data representation*, page 321.

### **Extended alignments (6.2.8)**

For information about extended alignments, see *data\_alignment*, page 359.

### **Valid alignments (6.2.8)**

For information about valid alignments on fundamental types, see the chapter *Data representation*.

**The value of the result of the sizeof operator (6.5.3.4)**

See *Data representation*, page 321.

**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. The chapter *Encodings* describes how to change the default encoding for the source character set, and by that the encoding for plain character constants and plain string literals in the execution character set.

**The meaning of the additional characters (5.2.1.2)**

Any multibyte characters in the extended source character set is translated into the following encoding for the execution character set:

| Execution character set | Encoding                             |
|-------------------------|--------------------------------------|
| <code>L</code> typed    | UTF-32                               |
| <code>u</code> typed    | UTF-16                               |
| <code>U</code> typed    | UTF-32                               |
| <code>u8</code> typed   | UTF-8                                |
| none typed              | The same as the source character set |

*Table 53: Translation of multibyte characters in the extended source 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)**

No shift states are supported.

**Direction of successive printing characters (5.2.2)**

The application defines the characteristics of a display device.

**The decimal point character (7.1.1)**

For a library with the configuration Normal or Tiny, the default decimal-point character is a '.'. For a library with the configuration Full, the chosen locale defines what character is used for the decimal point.

### **Printing characters (7.4, 7.30.2)**

The set of printing characters is determined by the chosen locale.

### **Control characters (7.4, 7.30.2)**

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, 7.30.2.1.2, 7.30.5.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11)**

The set of characters tested for the character-based functions are determined by the chosen locale. The set of characters tested for the `wchar_t`-based functions are the UTF-32 code points `0x0` to `0x7F`.

### **The native environment (7.11.1.1)**

The native environment is the same as the "C" locale.

### **Subject sequences for numeric conversion functions (7.22.1, 7.29.4.1)**

There are no additional subject sequences that can be accepted by the numeric conversion functions.

### **The collation of the execution character set (7.24.4.3, 7.29.4.4.2)**

Collation is not supported.

### Message returned by `strerror` (7.24.6.2)

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 54: Message returned by `strerror()`—DLIB runtime environment

### Formats for time and date (7.27.3.5, 7.29.5.1)

Time zone information is as you have implemented it in the low-level function `__getzone`.

### Character mappings (7.30.1)

The character mappings supported are `tolower` and `toupper`.

### Character classifications (7.30.1)

The character classifications that are supported are `alnum`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`.



# 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 523.

---

## 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 *System initialization*, page 144.

### 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. It is dependent on the chosen encoding for the source file. See *Text encodings*, page 244. By default, the source character set is Raw.

The execution character set is the set of legal characters that can appear in the execution environment. These are the execution character set for character constants and string literals and their encoding types:

| Execution character set | Encoding type            |
|-------------------------|--------------------------|
| L                       | UTF-32                   |
| u                       | UTF-16                   |
| U                       | UTF-32                   |
| u8                      | UTF-8                    |
| none                    | The source character set |

Table 55: Execution character sets and their encodings

The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 155.



### 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)

See *Locale*, page 155.

### 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 322, 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 326, 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 329, for information about *size\_t*.

### **Conversion from/to pointers (6.3.4)**

See *Casting*, page 329, for information about casting of data pointers and function pointers.

### **ptrdiff\_t (6.3.6, 7.1.1)**

See *ptrdiff\_t*, page 330, 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 322, 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
context_handler
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:** Some items in this list only apply when file descriptors are supported by the library configuration. For more information about runtime library configurations, see the chapter *The DLIB runtime environment*.

### **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 152 and *raise*, page 150, 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 122.

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 122.

**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 122.

**%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 147.

### `system()` (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *system*, page 152.

### 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 56: 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 153.

### `clock()` (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *clock*, page 146.



## A

- a (ielfdump option) . . . . . 473
- abort
  - implementation-defined behavior in C . . . . . 537
  - implementation-defined behavior in C89 (DLIB) . . . . . 552
  - system termination (DLIB) . . . . . 143
- absolute location
  - data, placing at (@) . . . . . 223
  - language support for . . . . . 185
  - #pragma location . . . . . 368
- addressing. *See* memory types, data models, and code models
- algorithm (library header file) . . . . . 397
- alias\_def (pragma directive) . . . . . 531
- alignment . . . . . 321
  - extended (implementation-defined behavior for C++) . 515
  - forcing stricter (#pragma data\_alignment) . . . . . 359
  - implementation-defined behavior for C++ . . . . . 508
  - in structures (#pragma pack) . . . . . 371
  - in structures, causing problems . . . . . 220
  - of an object (\_\_ALIGNOF\_\_) . . . . . 186
  - of data types . . . . . 322
  - restrictions for inline assembler . . . . . 162
- alignment (pragma directive) . . . . . 531, 549
- \_\_ALIGNOF\_\_ (operator) . . . . . 186
- alignof expression,
  - implementation-defined behavior for C++ . . . . . 508
- all (ielfdump option) . . . . . 474
- alternate\_target\_def (pragma directive) . . . . . 531
- anonymous structures . . . . . 220
- ANSI C. *See* C89
- application
  - building, overview of . . . . . 61
  - execution, overview of . . . . . 56
  - startup and termination (DLIB) . . . . . 141
- argv (argument), implementation-defined behavior in C. . 524
- array (library header file) . . . . . 397
- arrays
  - compiler padding . . . . . 322
  - global, accessing . . . . . 175
  - hints about index type . . . . . 219
  - implementation-defined behavior . . . . . 529
  - implementation-defined behavior in C89 . . . . . 547
  - non-lvalue . . . . . 188
  - of incomplete types . . . . . 187
    - single-value initialization . . . . . 189
- arrays of incomplete types . . . . . 203
- array::const\_iterator, implementation-defined behavior for C++ . . . . . 517
- array::iterator, implementation-defined behavior for C++ . 517
- asm, \_\_asm (language extension) . . . . . 161
  - implementation-defined behavior for C++ . . . . . 510
- assembler code
  - calling from C . . . . . 163
  - calling from C++ . . . . . 165
  - inserting inline . . . . . 161
- assembler directives
  - for call frame information . . . . . 177
  - using in inline assembler code . . . . . 162
- assembler instructions
  - CALLT . . . . . 340
  - inserting inline . . . . . 161
- assembler labels
  - default for application startup . . . . . 61, 109
  - making public (--public\_equ) . . . . . 283
  - prefixed by extra underscore . . . . . 294, 449
- assembler language interface . . . . . 159
  - calling convention. *See* assembler code
- assembler list file, generating . . . . . 270
- assembler output file . . . . . 164
- assembler statements . . . . . 190
- asserts
  - implementation-defined behavior of in C . . . . . 533
  - implementation-defined behavior of in C89, (DLIB) . . 550
  - including in application . . . . . 391
- assert.h (DLIB header file) . . . . . 396
- assignment of pointer types . . . . . 190

|                                                    |         |
|----------------------------------------------------|---------|
| @ (operator)                                       |         |
| placing at absolute address . . . . .              | 223     |
| placing in sections . . . . .                      | 224     |
| atexit limit, setting up . . . . .                 | 110     |
| atexit, reserving space for calls . . . . .        | 110     |
| atomic accesses . . . . .                          | 401     |
| atomic operations . . . . .                        | 80, 401 |
| __monitor . . . . .                                | 343     |
| atomic types for bitfields                         |         |
| implementation-defined behavior in C . . . . .     | 529     |
| atomic (library header file) . . . . .             | 397     |
| ATOMIC_..._LOCK_FREE macros,                       |         |
| implementation-defined behavior for C++ . . . . .  | 519     |
| attribute declaration,                             |         |
| implementation-defined behavior for C++ . . . . .  | 510     |
| attributes                                         |         |
| non-standard (implementation-                      |         |
| defined behavior for C++) . . . . .                | 510     |
| object . . . . .                                   | 338     |
| type . . . . .                                     | 335     |
| auto keyword . . . . .                             | 197     |
| auto variables . . . . .                           | 73      |
| at function entrance . . . . .                     | 168     |
| programming hints for efficient code . . . . .     | 232     |
| using in inline assembler code . . . . .           | 162     |
| auto, packing algorithm for initializers . . . . . | 419     |

## B

|                                                         |     |
|---------------------------------------------------------|-----|
| backtrace information <i>See</i> call frame information |     |
| bad_alloc::what,                                        |     |
| implementation-defined behavior for C++ . . . . .       | 513 |
| bad_array_new_length::what,                             |     |
| implementation-defined behavior for C++ . . . . .       | 513 |
| bad_cast::what,                                         |     |
| implementation-defined behavior for C++ . . . . .       | 514 |
| bad_exception::what,                                    |     |
| implementation-defined behavior for C++ . . . . .       | 514 |
| bad_typeid::what,                                       |     |
| implementation-defined behavior for C++ . . . . .       | 514 |

|                                                                |          |
|----------------------------------------------------------------|----------|
| bank (pragma directive) . . . . .                              | 355      |
| banks, switching . . . . .                                     | 355      |
| assembler language interface . . . . .                         | 167      |
| Barr, Michael . . . . .                                        | 39       |
| baseaddr (pragma directive) . . . . .                          | 531, 549 |
| __BASE_FILE__ (predefined symbol) . . . . .                    | 386      |
| basic_filebuf move                                             |          |
| constructor, implementation-defined behavior for C++ . . . . . | 518      |
| basic_filebuf::setbuf,                                         |          |
| implementation-defined behavior for C++ . . . . .              | 518      |
| basic_filebuf::sync,                                           |          |
| implementation-defined behavior for C++ . . . . .              | 518      |
| basic_ios::failure,                                            |          |
| implementation-defined behavior for C++ . . . . .              | 518      |
| basic_streambuf::setbuf,                                       |          |
| implementation-defined behavior for C++ . . . . .              | 518      |
| basic_stringbuf move                                           |          |
| constructor, implementation-defined behavior for C++ . . . . . | 518      |
| basic_template_matching (pragma directive) . . . . .           | 356      |
| using . . . . .                                                | 196      |
| batch files                                                    |          |
| error return codes . . . . .                                   | 243      |
| none for building library from command line . . . . .          | 130      |
| --bin (ielftool option) . . . . .                              | 474      |
| binary streams . . . . .                                       | 535      |
| binary streams in C89 (DLIB) . . . . .                         | 551      |
| --bin-multi (ielftool option) . . . . .                        | 475      |
| bit negation . . . . .                                         | 233      |
| bitfields                                                      |          |
| data representation of . . . . .                               | 324      |
| hints . . . . .                                                | 219      |
| implementation-defined behavior for C++ . . . . .              | 511      |
| implementation-defined behavior in C . . . . .                 | 529      |
| implementation-defined behavior in C89 . . . . .               | 547      |
| non-standard types in . . . . .                                | 186      |
| bitfields (pragma directive) . . . . .                         | 356      |
| bits in a byte, implementation-defined behavior in C . . . . . | 525      |
| bitset (library header file) . . . . .                         | 397      |
| bits, number of in                                             |          |
| one byte (implementation-defined behavior for C++) . . . . .   | 504      |

- bold style, in this guide . . . . . 40
  - bool (data type) . . . . . 322
    - adding support for in DLIB . . . . . 396, 400
  - \_\_break (intrinsic function) . . . . . 382
  - BRK (assembler instruction) . . . . . 382
  - .bss (ELF section) . . . . . 439
  - .bssf (ELF section) . . . . . 439
  - .bssf\_unit64kp (ELF section) . . . . . 437
  - .bssf.noinit (ELF section) . . . . . 440
  - .bss.noinit (ELF section) . . . . . 439
  - build for directive (in linker configuration file) . . . . . 407
  - building\_runtime (pragma directive) . . . . . 531, 549
  - \_\_BUILD\_NUMBER\_\_ (predefined symbol) . . . . . 386
  - bytes, number
  - of bits in (implementation-defined behavior for C++) . . . 504
- ## C
- C and C++ linkage . . . . . 167
  - C/C++ calling convention. *See* calling convention
  - C header files . . . . . 396
  - C language, overview . . . . . 183
  - call frame information . . . . . 177
    - in assembler list file . . . . . 164
    - in assembler list file (-IA) . . . . . 270
  - call frame information, disabling (--no\_call\_frame\_info) . 274
  - call graph root (stack usage control directive) . . . . . 450
  - call stack . . . . . 177
  - call table vectors, storage
  - callee-save registers, stored on stack. . . . . 73
  - calling convention
    - C++, requiring C linkage . . . . . 165
    - default (\_\_v2\_call) . . . . . 350
    - identifying . . . . . 386
    - in compiler . . . . . 166
    - overriding default (\_\_v1\_call) . . . . . 350
    - V1 . . . . . 166
    - V2 . . . . . 166
  - \_\_CALLING\_CONVENTION\_\_ (predefined symbol) . . . 386
  - calling\_convention (compiler option) . . . . . 256
  - calloc (library function) . . . . . 75
    - See also* heap
    - implementation-defined behavior in C89 (DLIB) . . . . 552
  - calls (pragma directive) . . . . . 357
  - \_\_callt (extended keyword) . . . . . 340
  - CALLT (assembler instruction) . . . . . 340
  - .callt0 (ELF section) . . . . . 440
  - call\_graph (linker option) . . . . . 294
  - call\_graph\_root (pragma directive) . . . . . 358
  - call-info (in stack usage control file) . . . . . 454
  - canaries . . . . . 85
  - can\_instantiate (pragma directive) . . . . . 531, 549
  - cassert (library header file) . . . . . 400
  - casting
    - implementation-defined behavior for C++ . . . . . 508–509
    - of pointers and integers . . . . . 329
    - pointers to integers, language extension . . . . . 188
  - category (in stack usage control file) . . . . . 453
  - ccomplex (library header file) . . . . . 400
  - cctype (DLIB header file) . . . . . 400
  - cerrno (DLIB header file) . . . . . 400
  - cexit (system termination code)
    - customizing system termination . . . . . 144
  - cfenv (library header file) . . . . . 400
  - CFI (assembler directive) . . . . . 177
  - cfloat (DLIB header file) . . . . . 400
  - char (data type) . . . . . 322
    - changing default representation (--char\_is\_signed) . . . 257
    - changing representation (--char\_is\_unsigned) . . . . . 257
    - implementation-defined behavior for C++ . . . . . 507
    - implementation-defined behavior in C . . . . . 526
    - signed and unsigned . . . . . 323
  - character literals,
    - implementation-defined behavior for C++ . . . . . 506, 511
  - character set
    - implementation-defined behavior . . . . . 504
    - implementation-defined behavior for C++ . . . . . 504
    - implementation-defined behavior in C . . . . . 524

|                                                       |          |
|-------------------------------------------------------|----------|
| characters                                            |          |
| implementation-defined behavior in C                  | 525      |
| implementation-defined behavior in C89                | 544      |
| --char_is_signed (compiler option)                    | 257      |
| --char_is_unsigned (compiler option)                  | 257      |
| char_traits<char16_t>::eof,                           |          |
| implementation-defined behavior for C++               | 515      |
| char_traits<char32_t>::eof,                           |          |
| implementation-defined behavior for C++               | 515      |
| char_traits<wchar_t>::eof,                            |          |
| implementation-defined behavior for C++               | 515      |
| char16_t (data type)                                  | 324      |
| implementation-defined behavior in C                  | 527      |
| char32_t (data type)                                  | 324      |
| implementation-defined behavior in C                  | 527      |
| check that (linker directive)                         | 431      |
| checksum                                              |          |
| calculation of                                        | 209      |
| display format in C-SPY for symbol                    | 217      |
| --checksum (ielftool option)                          | 475      |
| chrono (library header file)                          | 397      |
| cinttypes (DLIB header file)                          | 400      |
| ciso646 (library header file)                         | 400      |
| class memory (extended EC++)                          | 193      |
| class template partial specialization                 |          |
| matching (extended EC++)                              | 195      |
| class type, passing                                   |          |
| argument of (implementation-defined behavior for C++) | 509      |
| climits (DLIB header file)                            | 400      |
| locale (DLIB header file)                             | 400      |
| clock (DLIB library function),                        |          |
| implementation-defined behavior in C89                | 553      |
| clustering (compiler transformation)                  | 230      |
| disabling (--no_clustering)                           | 274      |
| cmath (DLIB header file)                              | 400      |
| code                                                  |          |
| execution of                                          | 63       |
| facilitating for good generation of                   | 231      |
| interruption of execution                             | 79       |
| --code (ielfdump option)                              | 480      |
| code models                                           | 77       |
| calling functions in                                  | 173      |
| configuration                                         | 63       |
| Far                                                   | 78       |
| identifying (__CODE_MODEL__)                          | 386      |
| Near                                                  | 78       |
| code motion (compiler transformation)                 | 229      |
| disabling (--no_code_motion)                          | 274      |
| codecvt (library header file)                         | 397      |
| codeseg (pragma directive)                            | 531, 549 |
| __CODE_MODEL__ (predefined symbol)                    | 386      |
| --code_model (compiler option)                        | 257, 260 |
| --code_section (compiler option)                      | 258      |
| --code_section (compiler option)                      | 258      |
| command line options                                  |          |
| <i>See also</i> compiler options                      |          |
| <i>See also</i> linker options                        |          |
| part of compiler invocation syntax                    | 239      |
| part of linker invocation syntax                      | 240      |
| passing                                               | 240      |
| typographic convention                                | 40       |
| command prompt icon, in this guide                    | 40       |
| .comment (ELF section)                                | 439      |
| comments                                              |          |
| after preprocessor directives                         | 188      |
| common block (call frame information)                 | 177      |
| common subexpr elimination (compiler transformation)  | 229      |
| disabling (--no_cse)                                  | 275      |
| compilation date                                      |          |
| exact time of (__TIME__)                              | 391      |
| identifying (__DATE__)                                | 387      |
| compiler                                              |          |
| environment variables                                 | 241      |
| invocation syntax                                     | 239      |
| output from                                           | 242      |
| compiler listing, generating (-l)                     | 270      |
| compiler object file                                  | 54       |
| including debug information in (--debug, -r)          | 260      |
| output from compiler                                  | 242      |

- compiler optimization levels . . . . . 227
- compiler options . . . . . 249
  - passing to compiler . . . . . 240
  - reading from file (-f) . . . . . 268
  - reading from file (--f) . . . . . 268
  - specifying parameters . . . . . 251
  - summary . . . . . 251
  - syntax . . . . . 249
    - for creating skeleton code . . . . . 164
    - instruction scheduling . . . . . 231
    - generate\_far\_runtime\_library\_calls . . . . . 269
    - warnings\_affect\_exit\_code . . . . . 243
- compiler platform, identifying . . . . . 388
- compiler subversion number . . . . . 390
- compiler transformations . . . . . 226
- compiler version number . . . . . 391
- compiling
  - from the command line . . . . . 61
  - syntax . . . . . 239
- complex (library header file) . . . . . 397
- complex.h (library header file) . . . . . 396
- computer style, typographic convention . . . . . 40
- concatenating strings . . . . . 190, 203
- concatenating wide string literals with different encoding types
  - implementation-defined behavior in C . . . . . 527
- condition\_variable (library header file) . . . . . 397
- config (linker option) . . . . . 295
- configuration
  - basic project settings . . . . . 61
  - \_\_low\_level\_init . . . . . 144
- configuration file for linker. *See* linker configuration file
- configuration symbols
  - for file input and output . . . . . 154
  - for strtod . . . . . 156
  - in library configuration files . . . . . 130
  - in linker configuration files . . . . . 432
  - specifying for linker . . . . . 295
- config\_def (linker option) . . . . . 295
- config\_search (linker option) . . . . . 295
- consistency, module . . . . . 117
- const
  - declaring objects . . . . . 334
  - .const (ELF section) . . . . . 440
  - constants, placing in named section . . . . . 359
  - .constf (ELF section) . . . . . 440
  - .consth (ELF section) . . . . . 441
  - constseg (pragma directive) . . . . . 358
  - contents, of this guide . . . . . 36
  - context\_handler (pragma directive) . . . . . 532, 549
  - control characters
    - implementation-defined behavior in C . . . . . 540
  - conventions, used in this guide . . . . . 39
  - copyright notice . . . . . 2
  - \_\_CORE\_\_ (predefined symbol) . . . . . 386
  - core
    - identifying . . . . . 386
    - RL78 0 (legacy name) . . . . . 47
    - RL78 1 (legacy name) . . . . . 47
    - RL78 2 (legacy name) . . . . . 47
    - selecting (--core) . . . . . 258
    - specifying on command line . . . . . 258
  - core (compiler option) . . . . . 258
  - cos (library function) . . . . . 394
  - cos (library routine) . . . . . 139–140
  - cosf (library routine) . . . . . 140
  - cosl (library routine) . . . . . 140
  - \_\_COUNTER\_\_ (predefined symbol) . . . . . 387
  - \_\_cplusplus (predefined symbol) . . . . . 387
  - cplusplus\_neutral (pragma directive) . . . . . 532
  - cpp\_init\_routine (linker option) . . . . . 296
  - create (iarchive option) . . . . . 480
  - cross call (compiler transformation) . . . . . 231
  - csetjmp (DLIB header file) . . . . . 400
  - csignal (DLIB header file) . . . . . 400
  - cspy\_support (pragma directive) . . . . . 532, 549
  - CSTACK (ELF block) . . . . . 441
    - See also* stack
    - setting up size for . . . . . 109

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| cstartup (system startup code)                                        |     |
| customizing system initialization . . . . .                           | 144 |
| source files for (DLIB) . . . . .                                     | 141 |
| cstat_disable (pragma directive) . . . . .                            | 353 |
| cstat_dump (pragma directive) . . . . .                               | 532 |
| cstat_enable (pragma directive) . . . . .                             | 353 |
| cstat_restore (pragma directive) . . . . .                            | 353 |
| cstat_suppress (pragma directive) . . . . .                           | 353 |
| cstdalign (DLIB header file) . . . . .                                | 400 |
| cstdarg (DLIB header file) . . . . .                                  | 400 |
| cstdbool (DLIB header file) . . . . .                                 | 400 |
| cstddef (DLIB header file) . . . . .                                  | 400 |
| cstdio (DLIB header file) . . . . .                                   | 400 |
| cstdlib (DLIB header file) . . . . .                                  | 400 |
| csdnoerror (DLIB header file) . . . . .                               | 400 |
| cstring (DLIB header file) . . . . .                                  | 400 |
| ctgmath (library header file) . . . . .                               | 400 |
| cthreads (DLIB header file) . . . . .                                 | 400 |
| ctime (DLIB header file) . . . . .                                    | 400 |
| cctype::table_size, implementation-defined behavior for C++ . . . . . | 516 |
| cctype.h (library header file) . . . . .                              | 396 |
| cuchar (DLIB header file) . . . . .                                   | 400 |
| cwctype.h (library header file) . . . . .                             | 400 |
| C_INCLUDE (environment variable) . . . . .                            | 241 |
| C-SPY                                                                 |     |
| debug support for C++ . . . . .                                       | 200 |
| interface to system termination . . . . .                             | 144 |
| C-STAT for static analysis, documentation for . . . . .               | 38  |
| C++                                                                   |     |
| absolute location . . . . .                                           | 224 |
| calling convention . . . . .                                          | 165 |
| header files . . . . .                                                | 397 |
| implementation-defined behavior . . . . .                             | 503 |
| language extensions . . . . .                                         | 201 |
| static member variables . . . . .                                     | 224 |
| support for . . . . .                                                 | 47  |
| --c++ (compiler option) . . . . .                                     | 259 |
| C++ header files . . . . .                                            | 397 |
| C++ objects, placing in memory type . . . . .                         | 71  |

|                                           |     |
|-------------------------------------------|-----|
| C++ terminology . . . . .                 | 39  |
| C++14. <i>See</i> Standard C++            |     |
| C18 standard . . . . .                    | 183 |
| C18. <i>See</i> Standard C                |     |
| C89                                       |     |
| implementation-defined behavior . . . . . | 543 |
| support for . . . . .                     | 183 |
| --c89 (compiler option) . . . . .         | 256 |

## D

|                                                       |               |
|-------------------------------------------------------|---------------|
| -D (compiler option) . . . . .                        | 259           |
| -d (iarchive option) . . . . .                        | 480           |
| data                                                  |               |
| alignment of . . . . .                                | 321           |
| different ways of storing . . . . .                   | 65            |
| located, declaring extern . . . . .                   | 224           |
| placing . . . . .                                     | 222, 359, 437 |
| at absolute location . . . . .                        | 223           |
| representation of . . . . .                           | 321           |
| storage . . . . .                                     | 65            |
| .data (ELF section) . . . . .                         | 441           |
| data block (call frame information) . . . . .         | 177           |
| data memory attributes, using . . . . .               | 67            |
| data models . . . . .                                 | 72            |
| configuration . . . . .                               | 62            |
| Far . . . . .                                         | 73            |
| Huge . . . . .                                        | 73            |
| identifying ( <code>__DATA_MODEL__</code> ) . . . . . | 387           |
| Near . . . . .                                        | 72            |
| data pointers . . . . .                               | 329           |
| data types . . . . .                                  | 322           |
| avoiding signed . . . . .                             | 219           |
| floating point . . . . .                              | 326           |
| in C++ . . . . .                                      | 334           |
| integer types . . . . .                               | 322           |
| .dataf (ELF section) . . . . .                        | 442           |
| .dataf_init (ELF section) . . . . .                   | 442           |
| dataseg (pragma directive) . . . . .                  | 359           |



- data\_alignment (pragma directive) . . . . . 359
- .data\_init (ELF section) . . . . . 441
- \_\_DATA\_MODEL\_\_ (predefined symbol) . . . . . 387
- .data\_unit64kp (ELF section) . . . . . 437
- \_\_DATE\_\_ (predefined symbol). . . . . 387
  - implementation-defined behavior for C++ . . . . . 512
- date (library function), configuring support for. . . . . 128
- DC32 (assembler directive) . . . . . 162
- debug (compiler option) . . . . . 260
- debug information, including in object file . . . . . 260
- .debug (ELF section). . . . . 438
- debug\_lib (linker option) . . . . . 296
- decimal point
  - implementation-defined behavior in C . . . . . 539
- declarations
  - empty . . . . . 189
  - Kernighan & Ritchie . . . . . 233
  - of functions . . . . . 167
- declarators, implementation-defined behavior in C89. . . . . 548
- decltype keyword . . . . . 197
- default\_to\_complex\_ranges (linker option) . . . . . 297
- define block (linker directive). . . . . 415
- define memory (linker directive) . . . . . 408
- define overlay (linker directive) . . . . . 417
- define region (linker directive) . . . . . 408
- define symbol (linker directive) . . . . . 432
- define\_symbol (linker option) . . . . . 297
- define\_type\_info (pragma directive). . . . . 532, 549
- delete (iarchive option). . . . . 480
- delete operator (extended EC++) . . . . . 198
- delete (keyword). . . . . 75
- denormalized numbers. *See* subnormal numbers
- dependencies (compiler option) . . . . . 260
- dependencies (linker option) . . . . . 297
- deprecated (pragma directive). . . . . 362
- deprecated\_feature\_warnings (compiler option). . . . . 262
- deque (library header file). . . . . 397
- destructors and interrupts, using . . . . . 199
- device description files, preconfigured for C-SPY . . . . . 48
- devices, interactive
  - implementation-defined behavior for C++ . . . . . 504
- DI (assembler instruction). . . . . 382
- diagnostic messages . . . . . 246
  - classifying as compilation errors . . . . . 262
  - classifying as compilation remarks . . . . . 263
  - classifying as compiler warnings . . . . . 263
  - classifying as errors . . . . . 275, 308
  - classifying as linker warnings . . . . . 300
  - classifying as linking errors . . . . . 298
  - classifying as linking remarks . . . . . 299
  - disabling compiler warnings . . . . . 279
  - disabling linker warnings . . . . . 311
  - disabling wrapping of in compiler . . . . . 279
  - disabling wrapping of in linker . . . . . 311
  - enabling compiler remarks . . . . . 284
  - enabling linker remarks . . . . . 314
  - listing all used by compiler . . . . . 264
  - listing all used by linker . . . . . 300
  - suppressing in compiler . . . . . 263
  - suppressing in linker . . . . . 299
- diagnostics
  - iarchive . . . . . 459
  - iobjmanip . . . . . 465
  - isymexport . . . . . 471
- diagnostics\_tables (compiler option) . . . . . 264
- diagnostics\_tables (linker option). . . . . 300
- diagnostics, implementation-defined behavior . . . . . 523
- diagnostics, implementation-defined behavior for C++ . . . . . 503
- diag\_default (pragma directive) . . . . . 363
- diag\_error (compiler option) . . . . . 262
- diag\_error (linker option) . . . . . 298
- no\_fragments (compiler option). . . . . 275
- no\_fragments (linker option) . . . . . 308
- diag\_error (pragma directive) . . . . . 363
- diag\_remark (compiler option). . . . . 263
- diag\_remark (linker option) . . . . . 299
- diag\_remark (pragma directive) . . . . . 364
- diag\_suppress (compiler option) . . . . . 263

|                                                                           |          |
|---------------------------------------------------------------------------|----------|
| --diag_suppress (linker option) . . . . .                                 | 299      |
| diag_suppress (pragma directive) . . . . .                                | 364      |
| --diag_warning (compiler option) . . . . .                                | 263      |
| --diag_warning (linker option) . . . . .                                  | 300      |
| diag_warning (pragma directive) . . . . .                                 | 364      |
| directives                                                                |          |
| pragma . . . . .                                                          | 49, 353  |
| to the linker . . . . .                                                   | 405      |
| directory, specifying as parameter . . . . .                              | 250      |
| --disable_div_mod_instructions (compiler option) . . . . .                | 264      |
| __disable_interrupt (intrinsic function) . . . . .                        | 382      |
| --disasm_data (ielfdump option) . . . . .                                 | 481      |
| --discard_unused_publics (compiler option) . . . . .                      | 264      |
| disclaimer . . . . .                                                      | 2        |
| DIVH instruction, disabling . . . . .                                     | 264      |
| DIVHU instruction, disabling . . . . .                                    | 264      |
| DLIB . . . . .                                                            | 395      |
| configurations . . . . .                                                  | 131      |
| configuring . . . . .                                                     | 129, 265 |
| naming convention . . . . .                                               | 41       |
| reference information. <i>See</i> the online help system . . . . .        | 393      |
| runtime environment . . . . .                                             | 121      |
| --dlib_config (compiler option) . . . . .                                 | 265      |
| DLib_Defaults.h (library configuration file) . . . . .                    | 130      |
| __DLIB_FILE_DESCRIPTOR (configuration symbol) . . . . .                   | 154      |
| do not initialize (linker directive) . . . . .                            | 421      |
| document conventions . . . . .                                            | 39       |
| documentation                                                             |          |
| contents of this . . . . .                                                | 36       |
| how to use this . . . . .                                                 | 35       |
| overview of guides . . . . .                                              | 37       |
| who should read this . . . . .                                            | 35       |
| \$\$ (in reserved identifiers) . . . . .                                  | 246      |
| domain errors, implementation-defined behavior in C . . . . .             | 534      |
| domain errors, implementation-defined behavior in C89<br>(DLIB) . . . . . | 550      |
| --double (compiler option) . . . . .                                      | 266      |
| double underscore (in reserved identifiers) . . . . .                     | 246      |
| double (data type) . . . . .                                              | 326      |
| avoiding . . . . .                                                        | 219      |

|                                                                         |          |
|-------------------------------------------------------------------------|----------|
| configuring size of floating-point type . . . . .                       | 63       |
| --do_explicit_zero_opt_in_named_sections<br>(compiler option) . . . . . | 266      |
| do_not_instantiate (pragma directive) . . . . .                         | 532, 549 |
| duplicate section merging (linker optimization) . . . . .               | 120      |
| dynamic initialization . . . . .                                        | 141      |
| and C++ . . . . .                                                       | 96       |
| dynamic memory . . . . .                                                | 75       |

## E

|                                                          |          |
|----------------------------------------------------------|----------|
| -e (compiler option) . . . . .                           | 267      |
| early_initialization (pragma directive) . . . . .        | 532, 549 |
| --edit (isymexport option) . . . . .                     | 481      |
| edition, of this guide . . . . .                         | 2        |
| EI (assembler instruction) . . . . .                     | 382      |
| ELF utilities . . . . .                                  | 457      |
| embedded systems, IAR special support for . . . . .      | 48       |
| empty region (in linker configuration file) . . . . .    | 413      |
| empty translation unit . . . . .                         | 190      |
| __enable_interrupt (intrinsic function) . . . . .        | 382      |
| --enable_restrict (compiler option) . . . . .            | 267      |
| enabling restrict keyword . . . . .                      | 267      |
| encodings . . . . .                                      | 244      |
| Raw . . . . .                                            | 244      |
| system default locale . . . . .                          | 245      |
| Unicode . . . . .                                        | 245      |
| UTF-16 . . . . .                                         | 245      |
| UTF-8 . . . . .                                          | 245      |
| --entry (linker option) . . . . .                        | 301      |
| entry label, program . . . . .                           | 141      |
| --entry_list_in_address_order (linker option) . . . . .  | 301      |
| entry, implementation-defined behavior for C++ . . . . . | 507      |
| enumerations                                             |          |
| implementation-defined behavior for C++ . . . . .        | 510      |
| implementation-defined behavior in C . . . . .           | 529      |
| implementation-defined behavior in C89 . . . . .         | 547      |
| enums                                                    |          |
| data representation . . . . .                            | 323      |

- forward declarations of . . . . . 188
- environment
  - implementation-defined behavior in C . . . . . 524
  - implementation-defined behavior in C89 . . . . . 543
  - runtime (DLIB) . . . . . 121
- environment names,
  - implementation-defined behavior in C . . . . . 525
- environment variables
  - C\_INCLUDE . . . . . 241
  - QCCRL78 . . . . . 241
- environment (native)
  - implementation-defined behavior in C . . . . . 540
- EQU (assembler directive) . . . . . 283
- ERANGE . . . . . 534
- ERANGE (C89) . . . . . 550
- errno value at underflow,
  - implementation-defined behavior in C . . . . . 537
- errno.h (library header file) . . . . . 396
- error messages . . . . . 247
  - classifying . . . . . 275, 308
  - classifying for compiler . . . . . 262
  - classifying for linker . . . . . 298
  - range . . . . . 115
- error return codes . . . . . 243
- error (linker directive) . . . . . 435
- error (pragma directive) . . . . . 365
- errors and warnings,
  - listing all used by the compiler (--diagnostics\_tables) . . . 264
- error\_category, implementation-defined behavior for C++ 513
- error\_limit (compiler option) . . . . . 267
- error\_limit (linker option) . . . . . 301
- escape sequences
  - implementation-defined behavior for C++ . . . . . 506
  - implementation-defined behavior in C . . . . . 526
- exception (library header file) . . . . . 398
- \_\_EXCEPTIONS\_\_ (predefined symbol) . . . . . 387
- exception\_neutral (pragma directive) . . . . . 532
- exception::what,
  - implementation-defined behavior for C++ . . . . . 514
- exclude (stack usage control directive) . . . . . 450
- execution character set,
  - implementation-defined behavior in C . . . . . 525
- execution character
  - set, implementation-defined behavior for C++ . . . . . 505
- execution wide-
  - character set, implementation-defined behavior for C++ . 505
- \_Exit (library function) . . . . . 143
- exit (library function) . . . . . 143
  - implementation-defined behavior for C++ . . . . . 513
  - implementation-defined behavior in C . . . . . 537
  - implementation-defined behavior in C89 . . . . . 553
- \_exit (library function) . . . . . 143
- \_\_exit (library function) . . . . . 143
- exp (library routine) . . . . . 139
- expf (library routine) . . . . . 140
- expl (library routine) . . . . . 140
- export (linker directive) . . . . . 433
- export\_builtin\_config (linker option) . . . . . 302
- export\_locals (isymexport option) . . . . . 481
- expressions (in linker configuration file) . . . . . 433
- extended alignment,
  - implementation-defined behavior for C++ . . . . . 515
- extended command line file
  - for compiler . . . . . 268
  - for linker . . . . . 302
  - passing options . . . . . 240
- extended keywords . . . . . 335
  - enabling (-e) . . . . . 267
  - overview . . . . . 49
  - summary . . . . . 339
  - syntax
    - object attributes . . . . . 339
    - type attributes on data objects . . . . . 68, 336
    - type attributes on functions . . . . . 337
- extended-selectors (in linker configuration file) . . . . . 430
- extern "C" linkage . . . . . 197
- extract (iarchive option) . . . . . 482

|                                                                    |               |
|--------------------------------------------------------------------|---------------|
| <b>F</b>                                                           |               |
| -f (compiler option) . . . . .                                     | 268           |
| -f (IAR utility option) . . . . .                                  | 482           |
| -f (linker option) . . . . .                                       | 302           |
| --f (compiler option) . . . . .                                    | 268           |
| --f (linker option) . . . . .                                      | 302           |
| --f (IAR utility option) . . . . .                                 | 483           |
| __far (extended keyword) . . . . .                                 | 341           |
| Far (code model) . . . . .                                         | 78            |
| Far (data model) . . . . .                                         | 73            |
| __far (extended keyword) . . . . .                                 | 329           |
| far (memory type) . . . . .                                        | 67            |
| __far_func (extended keyword) . . . . .                            | 341           |
| __far_func (function pointer) . . . . .                            | 328           |
| FAR_HEAP (section) . . . . .                                       | 442           |
| __FAR_RUNTIME_ATTRIBUTE__ (predefined symbol)                      | 387           |
| fatal error messages . . . . .                                     | 247           |
| fdopen, in stdio.h . . . . .                                       | 402           |
| fegetrapdisable . . . . .                                          | 402           |
| fegetrapenable . . . . .                                           | 402           |
| FENV_ACCESS, implementation-defined behavior in C                  | 528           |
| fenv.h (library header file) . . . . .                             | 396, 400      |
| additional C functionality . . . . .                               | 402           |
| fgetpos (library function)                                         |               |
| implementation-defined behavior in C . . . . .                     | 536           |
| implementation-defined behavior in C89 . . . . .                   | 552           |
| __FILE__ (predefined symbol) . . . . .                             | 388           |
| file buffering, implementation-defined behavior in C . . . . .     | 535           |
| file dependencies, tracking . . . . .                              | 260           |
| file input and output                                              |               |
| configuration symbols for . . . . .                                | 154           |
| file paths, specifying for #include files . . . . .                | 270           |
| file position, implementation-defined behavior in C . . . . .      | 535           |
| file (zero-length), implementation-defined behavior in C . . . . . | 535           |
| filename                                                           |               |
| extension for device description files . . . . .                   | 48            |
| extension for header files . . . . .                               | 48            |
| of object executable image . . . . .                               | 312           |
| of object file . . . . .                                           | 281, 312      |
| search procedure for . . . . .                                     | 241           |
| specifying as parameter . . . . .                                  | 250           |
| filenames (legal), implementation-defined behavior in C . . . . .  | 535           |
| fileno, in stdio.h . . . . .                                       | 402           |
| files, implementation-defined behavior in C                        |               |
| handling of temporary . . . . .                                    | 536           |
| multibyte characters in . . . . .                                  | 535           |
| opening . . . . .                                                  | 535           |
| --fill (ielftool option) . . . . .                                 | 484           |
| float (data type) . . . . .                                        | 326           |
| floating-point conversions                                         |               |
| implementation-defined behavior in C . . . . .                     | 528           |
| floating-point environment, accessing or not . . . . .             | 376           |
| floating-point expressions                                         |               |
| contracting or not . . . . .                                       | 376           |
| floating-point format . . . . .                                    | 326           |
| hints . . . . .                                                    | 219           |
| implementation-defined behavior in C . . . . .                     | 528           |
| implementation-defined behavior in C89 . . . . .                   | 546           |
| special cases . . . . .                                            | 328           |
| 32-bits . . . . .                                                  | 327           |
| 64-bits . . . . .                                                  | 327           |
| floating-point status flags . . . . .                              | 402           |
| floating-point type, configuring size of double . . . . .          | 63            |
| floating-point                                                     |               |
| conversion, implementation-defined behavior for C++ . . . . .      | 508           |
| floating-point                                                     |               |
| literals, implementation-defined behavior for C++ . . . . .        | 506           |
| floating-point                                                     |               |
| types, implementation-defined behavior for C++ . . . . .           | 508           |
| float.h (library header file) . . . . .                            | 396           |
| FLT_EVAL_METHOD, implementation-defined                            |               |
| behavior in C . . . . .                                            | 528, 533, 538 |
| FLT_ROUNDS, implementation-defined                                 |               |
| behavior in C . . . . .                                            | 528, 538      |
| fmod (library function),                                           |               |
| implementation-defined behavior in C89 . . . . .                   | 551           |
| --force_output (linker option) . . . . .                           | 303           |

- formats
    - floating-point values . . . . . 326
    - standard IEEE (floating point) . . . . . 326
  - forward\_list (library header file) . . . . . 398
  - FP\_CONTRACT, implementation-defined behavior in C . . . . . 529
  - fragmentation, of heap memory . . . . . 76
  - free (library function). *See also* heap . . . . . 75
  - freopen (function) . . . . . 404
  - front\_headers (ielftool option) . . . . . 484
  - fsetpos (library function), implementation-defined behavior in C . . . . . 536
  - fstream (library header file) . . . . . 398
  - ftell (library function)
    - implementation-defined behavior in C . . . . . 536
    - implementation-defined behavior in C89 . . . . . 552
  - Full DLIB (library configuration) . . . . . 132
  - \_\_func\_\_ (predefined symbol) . . . . . 388
    - implementation-defined behavior for C++ . . . . . 510
  - \_\_FUNCTION\_\_ (predefined symbol) . . . . . 388
  - function calls
    - calling convention . . . . . 166
    - eliminating overhead of by inlining . . . . . 83
    - Far code model . . . . . 174
    - Near code model . . . . . 173
    - preserved registers across. . . . . 168
  - function declarations, Kernighan & Ritchie . . . . . 233
  - function inlining . . . . . 120
  - function inlining (compiler transformation) . . . . . 229
    - disabling (--no\_inline) . . . . . 276
  - function names, prefixed by extra underscore. . . . . 294, 449
  - function pointer to object pointer conversion, implementation-defined behavior for C++ . . . . . 509
  - function pointers . . . . . 328
  - function prototypes . . . . . 232
    - enforcing . . . . . 284
  - function return addresses . . . . . 171
  - function table calls . . . . . 174
  - function template parameter deduction (extended EC++) . . . . . 196
  - function (pragma directive). . . . . 532, 549
  - function (stack usage control directive) . . . . . 450
  - functional (library header file) . . . . . 398
  - functions . . . . . 77
    - calling in different code models . . . . . 173
    - declaring . . . . . 167, 232
    - inlining. . . . . 229, 232, 366
    - interrupt . . . . . 79–80
    - intrinsic . . . . . 159, 232
    - monitor . . . . . 80
    - parameters . . . . . 168
    - placing in memory . . . . . 222, 224, 258
    - recursive
      - avoiding . . . . . 232
      - storing data on stack . . . . . 74
    - reentrancy (DLIB) . . . . . 394
    - related extensions. . . . . 77
    - return values from . . . . . 170
    - special function types. . . . . 79
  - function\_category (pragma directive) . . . . . 365, 532
  - function\_effects (pragma directive) . . . . . 532, 549
  - function-spec (in stack usage control file). . . . . 453
  - future (library header file). . . . . 398
- ## G
- g (ielfdump option) . . . . . 494
  - GCC attributes . . . . . 351
  - generate\_callt\_runtime\_library\_calls (compiler option) . . . . . 269
  - generate\_far\_runtime\_library\_calls (compiler option) . . . . . 269
  - generate\_vfe\_header (isymexport option) . . . . . 485
  - getw, in stdio.h . . . . . 402
  - getzone (library function), configuring support for. . . . . 128
  - \_\_get\_interrupt\_level (intrinsic function) . . . . . 382
  - \_\_get\_interrupt\_state (intrinsic function) . . . . . 382
  - get\_pointer\_safety, implementation-defined behavior for C++ . . . . . 514
  - global arrays, accessing . . . . . 175
  - global variables
    - accessing . . . . . 175

|                                             |     |
|---------------------------------------------|-----|
| affected by static clustering . . . . .     | 230 |
| handled during system termination . . . . . | 143 |
| hints for not using . . . . .               | 232 |
| initialized during system startup . . . . . | 142 |
| GRP_COMDAT, group type . . . . .            | 466 |
| --guard_calls (compiler option). . . . .    | 269 |
| guidelines, reading . . . . .               | 35  |

## H

|                                                    |          |
|----------------------------------------------------|----------|
| __halt (intrinsic function) . . . . .              | 383      |
| Harbison, Samuel P. . . . .                        | 39       |
| hardware multiplier/divider unit . . . . .         | 156      |
| hardware support in compiler . . . . .             | 121      |
| hash_map (library header file) . . . . .           | 398      |
| hash_set (library header file). . . . .            | 398      |
| .hbss (ELF section) . . . . .                      | 442      |
| .hbss.noinit (ELF section). . . . .                | 442      |
| .hdata (ELF section) . . . . .                     | 443      |
| .hdata_init (ELF section) . . . . .                | 443      |
| hdrstop (pragma directive) . . . . .               | 532, 549 |
| header files                                       |          |
| C . . . . .                                        | 396      |
| C++ . . . . .                                      | 397      |
| library . . . . .                                  | 393      |
| special function registers . . . . .               | 234      |
| DLib_Defaults.h . . . . .                          | 130      |
| implementation-defined behavior for C++ . . . . .  | 512      |
| including stdbool.h for bool . . . . .             | 323      |
| header names                                       |          |
| implementation-defined behavior in C . . . . .     | 530      |
| (implementation-defined behavior for C++ . . . . . | 505      |
| --header_context (compiler option). . . . .        | 270      |
| heap                                               |          |
| dynamic memory . . . . .                           | 75       |
| storing data . . . . .                             | 66       |
| VLA allocated on. . . . .                          | 289      |
| heap sections                                      |          |
| DLIB . . . . .                                     | 206      |

|                                                                  |     |
|------------------------------------------------------------------|-----|
| placing . . . . .                                                | 110 |
| heap size                                                        |     |
| and standard I/O. . . . .                                        | 207 |
| changing default. . . . .                                        | 110 |
| heap (zero-sized), implementation-defined behavior in C. . . . . | 537 |
| hide (isymexport directive) . . . . .                            | 469 |
| hints                                                            |     |
| for good code generation . . . . .                               | 231 |
| implementation-defined behavior . . . . .                        | 529 |
| using efficient data types . . . . .                             | 219 |
| __huge (extended keyword) . . . . .                              | 342 |
| Huge (data model) . . . . .                                      | 73  |
| __huge (extended keyword) . . . . .                              | 329 |
| HUGE_HEAP (section) . . . . .                                    | 443 |

## I

|                                                  |     |
|--------------------------------------------------|-----|
| -I (compiler option). . . . .                    | 270 |
| IAR Command Line Build Utility. . . . .          | 130 |
| IAR Systems Technical Support . . . . .          | 248 |
| iarbuild.exe (utility) . . . . .                 | 130 |
| iarhive . . . . .                                | 457 |
| commands summary . . . . .                       | 458 |
| options summary . . . . .                        | 459 |
| __iar_cos_accurate (library routine) . . . . .   | 140 |
| __iar_cos_accuratef (library routine) . . . . .  | 140 |
| __iar_cos_accuratef (library function) . . . . . | 394 |
| __iar_cos_accuratel (library routine) . . . . .  | 140 |
| __iar_cos_accuratel (library function) . . . . . | 394 |
| __iar_cos_small (library routine) . . . . .      | 139 |
| __iar_cos_smallf (library routine). . . . .      | 140 |
| __iar_cos_smallll (library routine). . . . .     | 140 |
| __iar_exp_small (library routine) . . . . .      | 139 |
| __iar_exp_smallf (library routine) . . . . .     | 140 |
| __iar_exp_smallll (library routine) . . . . .    | 140 |
| __iar_log_small (library routine) . . . . .      | 139 |
| __iar_log_smallf (library routine). . . . .      | 140 |
| __iar_log_smallll (library routine). . . . .     | 140 |
| __iar_log10_small (library routine) . . . . .    | 139 |

- \_\_iar\_log10\_smallf (library routine) . . . . . 140
- \_\_iar\_log10\_smalll (library routine) . . . . . 140
- \_\_iar\_maximum\_atexit\_calls . . . . . 110
- \_\_iar\_pow\_accurate (library routine) . . . . . 140
- \_\_iar\_pow\_accuratef (library routine) . . . . . 140
- \_\_iar\_pow\_accuratef (library function) . . . . . 394
- \_\_iar\_pow\_accuratel (library routine) . . . . . 140
- \_\_iar\_pow\_accuratel (library function) . . . . . 394
- \_\_iar\_pow\_small (library routine) . . . . . 139
- \_\_iar\_pow\_smallf (library routine) . . . . . 140
- \_\_iar\_pow\_smalll (library routine) . . . . . 140
- \_\_iar\_program\_start (label) . . . . . 141
- \_\_iar\_sin\_accurate (library routine) . . . . . 140
- \_\_iar\_sin\_accuratef (library routine) . . . . . 140
- \_\_iar\_sin\_accuratef (library function) . . . . . 394
- \_\_iar\_sin\_accuratel (library routine) . . . . . 140
- \_\_iar\_sin\_accuratel (library function) . . . . . 394
- \_\_iar\_sin\_small (library routine) . . . . . 139
- \_\_iar\_sin\_smallf (library routine) . . . . . 140
- \_\_iar\_sin\_smalll (library routine) . . . . . 140
- \_\_IAR\_SYSTEMS\_ICC\_\_ (predefined symbol) . . . . . 388
- \_\_iar\_tan\_accurate (library routine) . . . . . 140
- \_\_iar\_tan\_accuratef (library routine) . . . . . 140
- \_\_iar\_tan\_accuratef (library function) . . . . . 394
- \_\_iar\_tan\_accuratel (library routine) . . . . . 140
- \_\_iar\_tan\_accuratel (library function) . . . . . 394
- \_\_iar\_tan\_small (library routine) . . . . . 139
- \_\_iar\_tan\_smallf (library routine) . . . . . 140
- \_\_iar\_tan\_smalll (library routine) . . . . . 140
- \_\_iar\_tls.\$DATA (ELF section) . . . . . 444
- .iar.debug (ELF section) . . . . . 439
- .iar.dynexit (ELF section) . . . . . 443
- \_\_ICCRL78\_\_ (predefined symbol) . . . . . 388
- icons
  - in this guide . . . . . 40
- IDE
  - building a library from . . . . . 130
  - overview of build tools . . . . . 45
- ident (pragma directive) . . . . . 532
- identifiers
  - implementation-defined behavior in C . . . . . 525
  - implementation-defined behavior in C89 . . . . . 544
  - reserved . . . . . 246
- IEEE format, floating-point values . . . . . 326
- ielfdump . . . . . 462
  - options summary . . . . . 463
- ielftool . . . . . 460
  - options summary . . . . . 461
- if (linker directive) . . . . . 436
- ihex (ielftool option) . . . . . 485
- ILINK options. *See* linker options
- ILINK. *See* linker
- image\_input (linker option) . . . . . 303
- implements\_aspect (pragma directive) . . . . . 532
- important\_typedef (pragma directive) . . . . . 532, 549
- #include directive,
  - implementation-defined behavior for C++ . . . . . 512
- include files
  - including before source files . . . . . 282
  - search procedure implementation for C++ . . . . . 512
  - specifying . . . . . 241
- include (linker directive) . . . . . 436
- include\_alias (pragma directive) . . . . . 366
- infinity . . . . . 328
- infinity (style for printing), implementation-defined
  - behavior in C . . . . . 536
- initialization
  - changing default . . . . . 110
  - C++ dynamic . . . . . 96
  - dynamic . . . . . 141
  - manual . . . . . 111
  - packing algorithm for . . . . . 111
  - single-value . . . . . 189
  - suppressing . . . . . 110
- initialization\_routine (pragma directive) . . . . . 532
- initialize (linker directive) . . . . . 418
- initializers, static . . . . . 188
- initializer\_list (library header file) . . . . . 398
- .init\_array (section) . . . . . 444

|                                                                         |          |                                                          |     |
|-------------------------------------------------------------------------|----------|----------------------------------------------------------|-----|
| init_routines_only_for_needed_variables<br>(pragma directive) . . . . . | 532      | interrupt state, restoring . . . . .                     | 384 |
| --inline (linker option) . . . . .                                      | 304      | interrupt vector . . . . .                               | 79  |
| inline assembler . . . . .                                              | 161      | specifying with pragma directive . . . . .               | 378 |
| avoiding . . . . .                                                      | 232      | interrupt vector table                                   |     |
| <i>See also</i> assembler language interface                            |          | start address for . . . . .                              | 79  |
| inline functions                                                        |          | .intvec section . . . . .                                | 444 |
| in compiler . . . . .                                                   | 229      | interruptions                                            |     |
| inline (pragma directive) . . . . .                                     | 366      | disabling . . . . .                                      | 343 |
| inline_template (pragma directive) . . . . .                            | 532      | during function execution . . . . .                      | 80  |
| inlining . . . . .                                                      | 120      | processor state . . . . .                                | 73  |
| inlining functions . . . . .                                            | 83       | using with C++ destructors . . . . .                     | 199 |
| implementation-defined behavior . . . . .                               | 529      | intptr_t (integer type) . . . . .                        | 330 |
| installation directory . . . . .                                        | 39       | __intrinsic (extended keyword) . . . . .                 | 343 |
| instantiate (pragma directive) . . . . .                                | 532, 549 | intrinsic functions . . . . .                            | 232 |
| instruction scheduling (compiler option) . . . . .                      | 231      | overview . . . . .                                       | 159 |
| int (data type) signed and unsigned . . . . .                           | 323      | summary . . . . .                                        | 381 |
| integer to floating-point conversion,                                   |          | intrinsics.h (header file) . . . . .                     | 381 |
| implementation-defined behavior for C++ . . . . .                       | 508      | inttypes.h (library header file) . . . . .               | 396 |
| integer to pointer                                                      |          | .intvec (section) . . . . .                              | 444 |
| conversion, implementation-defined behavior for C++ . . . . .           | 509      | invocation syntax . . . . .                              | 239 |
| integer types . . . . .                                                 | 322      | iobjmanip . . . . .                                      | 464 |
| casting . . . . .                                                       | 329      | options summary . . . . .                                | 465 |
| implementation-defined behavior . . . . .                               | 527      | iomanip (library header file) . . . . .                  | 398 |
| implementation-defined behavior for C++ . . . . .                       | 507, 509 | ios (library header file) . . . . .                      | 398 |
| implementation-defined behavior in C89 . . . . .                        | 545      | iosfwd (library header file) . . . . .                   | 398 |
| intptr_t . . . . .                                                      | 330      | iostream classes                                         |     |
| ptrdiff_t . . . . .                                                     | 330      | implementation-defined behavior for C++ . . . . .        | 517 |
| size_t . . . . .                                                        | 329      | iostream (library header file) . . . . .                 | 398 |
| uintptr_t . . . . .                                                     | 330      | ios_base::streamoff, implementation-defined behavior for |     |
| integral promotion . . . . .                                            | 233      | C++ . . . . .                                            | 519 |
| Intel hex . . . . .                                                     | 205      | ios_base::streampos, implementation-defined behavior for |     |
| interactive devices                                                     |          | C++ . . . . .                                            | 519 |
| implementation-defined behavior for C++ . . . . .                       | 504      | ios_base::sync_with_stdio,                               |     |
| internal error . . . . .                                                | 248      | implementation-defined behavior for C++ . . . . .        | 518 |
| __interrupt (extended keyword) . . . . .                                | 80, 342  | iso646.h (library header file) . . . . .                 | 396 |
| using in pragma directives . . . . .                                    | 378      | istream (library header file) . . . . .                  | 398 |
| interrupt functions . . . . .                                           | 79       | iswalnum (function) . . . . .                            | 404 |
| interrupt handler. <i>See</i> interrupt service routine                 |          | iswxdigit (function) . . . . .                           | 404 |
| interrupt service routine . . . . .                                     | 79       | ismexport . . . . .                                      | 467 |
|                                                                         |          | options summary . . . . .                                | 468 |



italic style, in this guide . . . . . 40  
 iterator (library header file) . . . . . 398  
 I/O register. *See* SFR

## K

--keep (linker option) . . . . . 304  
 keep (linker directive) . . . . . 421  
 keep\_definition (pragma directive) . . . . . 532, 549  
 Kernighan & Ritchie function declarations . . . . . 233  
   disallowing . . . . . 284  
 keywords . . . . . 335  
   extended, overview of . . . . . 49

## L

-L (linker option) . . . . . 314  
 -l (compiler option) . . . . . 270  
   for creating skeleton code . . . . . 164  
 labels . . . . . 189  
   assembler, making public . . . . . 283  
   assembler, prefixed by extra underscore . . . . . 294, 449  
   \_\_iar\_program\_start . . . . . 141  
   \_\_program\_start . . . . . 141  
 Labrosse, Jean J. . . . . 39  
 language extensions  
   enabling using pragma . . . . . 367  
   enabling (-e) . . . . . 267  
 language overview . . . . . 47  
 language (pragma directive) . . . . . 367  
 libraries  
   reason for using . . . . . 54  
   using a prebuilt . . . . . 132  
 libraries, required  
 (implementation-defined behavior for C++) . . . . . 503  
 library configuration files  
   DLIB . . . . . 131  
   DLib\_Defaults.h . . . . . 130  
   modifying . . . . . 130

  specifying . . . . . 265  
 library documentation . . . . . 393  
 library files, linker search path to (--search) . . . . . 314  
 library functions  
   summary, DLIB . . . . . 396  
   online help for . . . . . 38  
 library header files . . . . . 393  
 library modules  
   introduction . . . . . 88  
   overriding . . . . . 128  
 library object files . . . . . 394  
 library project, building using a template . . . . . 130  
 library\_default\_requirements (pragma directive) . . . . . 532, 549  
 library\_provides (pragma directive) . . . . . 532, 550  
 library\_requirement\_override (pragma directive) . . . . . 532, 550  
 lightbulb icon, in this guide . . . . . 40  
 limits (library header file) . . . . . 398  
 limits.h (library header file) . . . . . 396  
 \_\_LINE\_\_ (predefined symbol) . . . . . 389  
 linkage, C and C++ . . . . . 167  
   implementation-defined behavior for C++ . . . . . 510–511, 513  
 linker . . . . . 87  
   checking section types when linking . . . . . 407  
   output from . . . . . 244  
 linker configuration file  
   for placing code and data . . . . . 91  
   in depth . . . . . 405, 449  
   overview of . . . . . 405, 449  
   selecting . . . . . 105  
 linker object executable image  
 specifying filename of (-o) . . . . . 312  
 linker optimizations . . . . . 119  
   duplicate section merging . . . . . 120  
   small function inlining . . . . . 120  
   virtual function elimination . . . . . 119  
 linker options . . . . . 291  
   reading from file (-f) . . . . . 302  
   reading from file (--f) . . . . . 302  
   summary . . . . . 291  
   typographic convention . . . . . 40

|                                                        |          |
|--------------------------------------------------------|----------|
| linking                                                |          |
| from the command line                                  | 61       |
| in the build process                                   | 55       |
| introduction                                           | 87       |
| process for                                            | 89       |
| list (library header file)                             | 398      |
| listing, generating                                    | 270      |
| literature, recommended                                | 39       |
| local symbols, removing from ELF image                 | 310      |
| local variables, <i>See</i> auto variables             |          |
| locale                                                 |          |
| changing at runtime                                    | 155      |
| implementation-defined behavior for C++                | 516      |
| implementation-defined behavior in C                   | 527, 539 |
| library header file                                    | 398      |
| support for                                            | 155      |
| locale object, implementation-defined behavior for C++ | 516      |
| locale.h (library header file)                         | 396      |
| located data, declaring extern                         | 224      |
| location (pragma directive)                            | 223, 368 |
| --log (linker option)                                  | 305      |
| log (library routine)                                  | 139      |
| logf (library routine)                                 | 140      |
| logical (linker directive)                             | 409      |
| logl (library routine)                                 | 140      |
| --log_file (linker option)                             | 306      |
| log10 (library routine)                                | 139      |
| log10f (library routine)                               | 140      |
| log10l (library routine)                               | 140      |
| long double (data type)                                | 326      |
| long float (data type), synonym for double             | 188      |
| long long (data type)                                  |          |
| avoiding                                               | 219      |
| long long (data type) signed and unsigned              | 323      |
| long (data type) signed and unsigned                   | 323      |
| longjmp, restrictions for using                        | 395      |
| loop unrolling (compiler transformation)               | 229      |
| disabling                                              | 279      |
| #pragma unroll                                         | 377      |

|                                                     |          |
|-----------------------------------------------------|----------|
| loop-invariant expressions                          | 229      |
| __low_level_init                                    | 141      |
| customizing                                         | 144      |
| initialization phase                                | 57       |
| low_level_init.c                                    | 141      |
| low-level processor operations                      | 184, 381 |
| accessing                                           | 159      |
| lvalue-to-rvalue                                    |          |
| conversion, implementation-defined behavior for C++ | 508      |
| lz77, packing algorithm for initializers            | 419      |

## M

|                                                    |          |
|----------------------------------------------------|----------|
| __mach (intrinsic function)                        | 383      |
| MACH (assembler instruction), inserting            | 383      |
| __machu (intrinsic function)                       | 383      |
| MACHU (assembler instruction), inserting           | 383      |
| macros                                             |          |
| embedded in #pragma optimize                       | 371      |
| ERANGE (in errno.h)                                | 534, 550 |
| inclusion of assert                                | 391      |
| NULL, implementation-defined behavior              |          |
| in C89 for DLIB                                    | 550      |
| NULL, implementation-defined behavior in C         | 534      |
| substituted in #pragma directives                  | 184      |
| --macro_positions_in_diagnostics (compiler option) | 271      |
| main (function)                                    |          |
| definition (C89)                                   | 543      |
| implementation-defined behavior for C++            | 506–507  |
| implementation-defined behavior in C               | 524      |
| malloc (library function)                          |          |
| <i>See also</i> heap                               | 75       |
| implementation-defined behavior in C89             | 552      |
| --mangled_names_in_messages (linker option)        | 306      |
| Mann, Bernhard                                     | 39       |
| --manual_dynamic_initialization (linker option)    | 306      |
| -map (linker option)                               | 307      |
| map file, producing                                | 307      |
| map (library header file)                          | 398      |

- math functions rounding mode, implementation-defined behavior in C . . . . . 538
- math functions (library functions). . . . . 139
- math.h (library header file). . . . . 396
- max recursion depth (stack usage control directive) . . . . 451
- max\_cost\_constexpr\_call (compiler option). . . . . 272
- max\_depth\_constexpr\_call (compiler option) . . . . . 272
- MB\_LEN\_MAX, implementation-defined behavior in C . 538
- member functions, pointers to. . . . . 200
- memory
  - accessing . . . . . 66, 174
    - using data16 method . . . . . 175–176
    - using saddr method . . . . . 175
  - allocating in C++. . . . . 75
  - dynamic . . . . . 75
  - heap . . . . . 75
  - non-initialized . . . . . 235
  - RAM, saving . . . . . 232
  - releasing in C++. . . . . 75
  - stack. . . . . 73
    - saving . . . . . 232
  - used by global or static variables . . . . . 65
- memory map
  - initializing SFRs . . . . . 144
  - linker configuration for . . . . . 106
  - output from linker . . . . . 244
  - producing (--map) . . . . . 307
- memory placement
  - using type definitions. . . . . 69
- memory types . . . . . 66
  - C++. . . . . 71
  - placing variables in . . . . . 71
  - pointers . . . . . 69
  - SFR . . . . . 67
  - specifying . . . . . 67
  - structures . . . . . 70
  - summary . . . . . 67
- memory (library header file). . . . . 398
- memory (pragma directive). . . . . 532, 550
- \_\_memory\_of
  - operator . . . . . 194
- merge duplicate sections. . . . . 120
- merge\_duplicate\_sections (linker option) . . . . . 307
- message (pragma directive) . . . . . 368
- messages
  - disabling . . . . . 284, 315
  - forcing . . . . . 368
- messages::do\_close, implementation-defined behavior for C++ . . . . . 516
- messages::do\_get, implementation-defined behavior for C++ . . . . . 516
- messages::do\_open, implementation-defined behavior for C++ . . . . . 516
- Meyers, Scott . . . . . 39
- mfc (compiler option). . . . . 272
- migration
  - from a UBROF-based product . . . . . 38
- MISRA C
  - documentation . . . . . 38
- misrac\_verbose (compiler option) . . . . . 253
- misrac\_verbose (linker option). . . . . 292
- misrac1998 (compiler option) . . . . . 253
- misrac1998 (linker option). . . . . 292
- misrac2004 (compiler option) . . . . . 253
- misrac2004 (linker option). . . . . 292
- mode changing, implementation-defined behavior in C . . 536
- module consistency. . . . . 117
  - rtmodel. . . . . 373
- modules, introduction . . . . . 88
- module\_name (pragma directive) . . . . . 532, 550
- module-spec (in stack usage control file) . . . . . 453
- \_\_monitor (extended keyword). . . . . 343
- monitor functions . . . . . 80, 343
- Motorola S-records. . . . . 205
- multibyte characters
  - implementation-defined behavior for C++. . . . . 515
  - implementation-defined behavior in C. . . . . 525, 539
- multithreaded environment . . . . . 157

|                                                              |     |
|--------------------------------------------------------------|-----|
| multi-character literals,                                    |     |
| value of (implementation-defined behavior for C++) . . . . . | 505 |
| multi-file compilation . . . . .                             | 226 |
| multi-threaded environment                                   |     |
| implementation-defined behavior in C . . . . .               | 524 |
| mutex (library header file) . . . . .                        | 398 |

## N

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| name (in stack usage control file) . . . . .                          | 454 |
| names block (call frame information) . . . . .                        | 177 |
| naming conventions . . . . .                                          | 41  |
| NaN                                                                   |     |
| implementation of . . . . .                                           | 328 |
| implementation-defined behavior in C . . . . .                        | 536 |
| native environment                                                    |     |
| implementation-defined behavior in C . . . . .                        | 540 |
| native_handle_type, implementation-defined behavior for C++ . . . . . | 519 |
| native_handle, implementation-defined behavior for C++ . . . . .      | 519 |
| NDEBUG (preprocessor symbol) . . . . .                                | 391 |
| __near (extended keyword) . . . . .                                   | 343 |
| Near (code model) . . . . .                                           | 78  |
| Near (data model) . . . . .                                           | 72  |
| __near (extended keyword) . . . . .                                   | 329 |
| near (memory type) . . . . .                                          | 67  |
| --near_const_location (compiler option) . . . . .                     | 273 |
| __near_func (extended keyword) . . . . .                              | 344 |
| __near_func (function pointer) . . . . .                              | 328 |
| NEAR_HEAP (section) . . . . .                                         | 444 |
| negative values,                                                      |     |
| right shifting (implementation-defined behavior for C++) . . . . .    | 509 |
| new operator (extended EC++) . . . . .                                | 198 |
| new (keyword) . . . . .                                               | 75  |
| new (library header file) . . . . .                                   | 398 |
| no calls from (stack usage control directive) . . . . .               | 452 |
| --nonportable_path_warnings (compiler option) . . . . .               | 279 |
| non-initialized variables, hints for . . . . .                        | 236 |
| Non-Plain Old Functions (POF),                                        |     |
| implementation-defined behavior for C++ . . . . .                     | 514 |

|                                                          |          |
|----------------------------------------------------------|----------|
| non-scalar parameters, avoiding . . . . .                | 232      |
| __noreturn (extended keyword) . . . . .                  | 346      |
| Normal DLIB (library configuration) . . . . .            | 132      |
| Not a number (NaN) . . . . .                             | 328      |
| not-separated (Renesas CC-RL compiler section) . . . . . | 115      |
| __no_alloc (extended keyword) . . . . .                  | 344      |
| __no_alloc_str (operator) . . . . .                      | 345      |
| __no_alloc_str16 (operator) . . . . .                    | 345      |
| __no_alloc16 (extended keyword) . . . . .                | 344      |
| __no_bit_access (extended keyword) . . . . .             | 345      |
| --no_bom (ielfdump option) . . . . .                     | 485      |
| --no_bom (iobjmanip option) . . . . .                    | 485      |
| --no_bom (isymexport option) . . . . .                   | 485      |
| --no_bom (compiler option) . . . . .                     | 273      |
| --no_bom (iarchive option) . . . . .                     | 485      |
| --no_bom (linker option) . . . . .                       | 308      |
| --no_call_frame_info (compiler option) . . . . .         | 274      |
| --no_clustering (compiler option) . . . . .              | 274      |
| --no_code_motion (compiler option) . . . . .             | 274      |
| --no_cross_call (compiler option) . . . . .              | 275      |
| --no_cse (compiler option) . . . . .                     | 275      |
| --no_entry (linker option) . . . . .                     | 308      |
| --no_free_heap (linker option) . . . . .                 | 309      |
| --no_header (ielfdump option) . . . . .                  | 486      |
| __no_init (extended keyword) . . . . .                   | 236, 346 |
| --no_inline (compiler option) . . . . .                  | 276      |
| --no_inline (linker option) . . . . .                    | 309      |
| --no_library_search (linker option) . . . . .            | 309      |
| --no_locals (linker option) . . . . .                    | 310      |
| __no_operation (intrinsic function) . . . . .            | 383      |
| --no_path_in_file_macros (compiler option) . . . . .     | 276      |
| no_pch (pragma directive) . . . . .                      | 532, 550 |
| --no_range_reservations (linker option) . . . . .        | 310      |
| --no_rel_section (ielfdump option) . . . . .             | 486      |
| --no_remove (linker option) . . . . .                    | 310      |
| __no_save (extended keyword) . . . . .                   | 346      |
| --no_scheduling (compiler option) . . . . .              | 276      |
| --no_size_constraints (compiler option) . . . . .        | 276      |
| no_stack_protect (pragma directive) . . . . .            | 369      |

- no\_static\_destruction (compiler option) . . . . . 277
  - no\_strtab (ielfdump option) . . . . . 486
  - no\_system\_include (compiler option) . . . . . 277
  - no\_tbaa (compiler option) . . . . . 277
  - no\_typedefs\_in\_diagnostics (compiler option) . . . . . 278
  - no\_uniform\_attribute\_syntax (compiler option) . . . . . 278
  - no\_unroll (compiler option) . . . . . 279
  - no\_utf8\_in (ielfdump option) . . . . . 487
  - no\_vfe (linker option) . . . . . 310
  - no\_vtable\_use (pragma directive) . . . . . 532
  - no\_warnings (compiler option) . . . . . 279
  - no\_warnings (linker option) . . . . . 311
  - no\_workseg (pragma directive) . . . . . 369
  - no\_wrap\_diagnostics (compiler option) . . . . . 279
  - no\_wrap\_diagnostics (linker option) . . . . . 311
  - NULL
    - implementation-defined behavior for C++ . . . . . 513
    - implementation-defined behavior in C . . . . . 534
    - implementation-defined behavior in C89 (DLIB) . . . . . 550
    - pointer constant, relaxation to Standard C . . . . . 188
  - numbers (in linker configuration file) . . . . . 434
  - numeric conversion functions
    - implementation-defined behavior in C . . . . . 540
  - numeric (library header file) . . . . . 398
- O**
- O (compiler option) . . . . . 280
  - o (compiler option) . . . . . 281
  - o (iarchive option) . . . . . 488
  - o (ielfdump option) . . . . . 488
  - o (linker option) . . . . . 312
  - object attributes . . . . . 338
  - object filename, specifying (-o) . . . . . 281, 312
  - object files, linker search path to (--search) . . . . . 314
  - object pointer to function pointer conversion, implementation-defined behavior for C++ . . . . . 509
  - object\_attribute (pragma directive) . . . . . 236, 369
  - offset (ielftool option) . . . . . 487
  - once (pragma directive) . . . . . 532, 550
  - only\_stdout (compiler option) . . . . . 280
  - only\_stdout (linker option) . . . . . 311
  - open\_s (function) . . . . . 404
  - operators
    - See also* @ (operator)
    - for region expressions . . . . . 413
    - for section control . . . . . 186
    - in inline assembler . . . . . 161
    - new and delete . . . . . 198
    - precision for 32-bit float . . . . . 327
    - precision for 64-bit float . . . . . 327
    - sizeof, implementation-defined behavior in C . . . . . 539
    - \_\_ALIGNOF\_\_, for alignment control . . . . . 186
    - \_\_memory\_of\_\_ . . . . . 194
    - ?, language extensions for . . . . . 201
  - optimization
    - clustering, disabling . . . . . 274
    - code motion, disabling . . . . . 274
    - common sub-expression elimination, disabling . . . . . 275
    - configuration . . . . . 63
    - disabling . . . . . 228
    - function inlining, disabling (--no\_inline) . . . . . 276
    - hints . . . . . 231
    - loop unrolling, disabling . . . . . 279
    - scheduling, disabling . . . . . 276
    - specifying (-O) . . . . . 280
    - techniques . . . . . 228
    - type-based alias analysis, disabling (--tbaa) . . . . . 277
    - using inline assembler code . . . . . 162
    - using pragma directive . . . . . 370
  - optimization levels . . . . . 227
  - optimize (pragma directive) . . . . . 370
  - option parameters . . . . . 249
  - options, compiler. *See* compiler options
  - options, iarchive. *See* iarchive options
  - options, ielfdump. *See* ielfdump options
  - options, ielftool. *See* ielftool options
  - options, iobjmanip. *See* iobjmanip options

|                                                    |          |
|----------------------------------------------------|----------|
| options, isymexport. <i>See</i> isymexport options |          |
| options, linker. <i>See</i> linker options         |          |
| .option_byte (section) . . . . .                   | 444      |
| --option_name (compiler option) . . . . .          | 300      |
| Oram, Andy . . . . .                               | 39       |
| ostream (library header file) . . . . .            | 398      |
| output                                             |          |
| from preprocessor . . . . .                        | 282      |
| specifying for linker. . . . .                     | 61       |
| --output (compiler option). . . . .                | 281      |
| --output (iarchive option) . . . . .               | 488      |
| --output (ielfdump option) . . . . .               | 488      |
| --output (linker option) . . . . .                 | 312      |
| overhead, reducing . . . . .                       | 229      |
| over-aligned types,                                |          |
| implementation-defined behavior for C++ . . . . .  | 509, 514 |

## P

|                                                        |          |
|--------------------------------------------------------|----------|
| pack (pragma directive) . . . . .                      | 331, 371 |
| packbits, packing algorithm for initializers . . . . . | 419      |
| __packed (extended keyword) . . . . .                  | 347      |
| packed structure types. . . . .                        | 331      |
| packing, algorithms for initializers . . . . .         | 419      |
| parameters                                             |          |
| function . . . . .                                     | 168      |
| hidden . . . . .                                       | 169      |
| non-scalar, avoiding. . . . .                          | 232      |
| register. . . . .                                      | 168–169  |
| rules for specifying a file or directory . . . . .     | 250      |
| specifying . . . . .                                   | 251      |
| stack. . . . .                                         | 168, 170 |
| typographic convention . . . . .                       | 40       |
| --parity (ielftool option) . . . . .                   | 488      |
| part number, of this guide . . . . .                   | 2        |
| --pending_instantiations (compiler option). . . . .    | 281      |
| permanent registers. . . . .                           | 168      |
| error (library function),                              |          |
| implementation-defined behavior in C89 . . . . .       | 552      |

|                                                                  |          |
|------------------------------------------------------------------|----------|
| place at (linker directive) . . . . .                            | 422      |
| place in (linker directive) . . . . .                            | 423      |
| placeholder objects,                                             |          |
| implementation-defined behavior for C++ . . . . .                | 514      |
| placement                                                        |          |
| code and data . . . . .                                          | 437      |
| in named sections. . . . .                                       | 224      |
| of code and data, introduction to . . . . .                      | 91       |
| --place_holder (linker option). . . . .                          | 312      |
| plain char                                                       |          |
| implementation-defined behavior for C++ . . . . .                | 507      |
| implementation-defined behavior in C. . . . .                    | 526      |
| pointer safety,                                                  |          |
| implementation-defined behavior for C++ . . . . .                | 507, 514 |
| pointer to integer                                               |          |
| conversion, implementation-defined behavior for C++ . . . . .    | 509      |
| pointer types . . . . .                                          | 328      |
| differences between . . . . .                                    | 70       |
| mixing . . . . .                                                 | 188      |
| pointer types, implementation-defined behavior for C++ . . . . . | 508      |
| pointers                                                         |          |
| casting . . . . .                                                | 329      |
| data . . . . .                                                   | 329      |
| function . . . . .                                               | 328      |
| implementation-defined behavior . . . . .                        | 529      |
| implementation-defined behavior for C++ . . . . .                | 507      |
| implementation-defined behavior in C89 . . . . .                 | 547      |
| pointers to different function types . . . . .                   | 190      |
| pointer_safety::preferred,                                       |          |
| implementation-defined behavior for C++ . . . . .                | 514      |
| pointer_safety::relaxed,                                         |          |
| implementation-defined behavior for C++ . . . . .                | 514      |
| pop_macro (pragma directive) . . . . .                           | 532      |
| porting, code containing pragma directives. . . . .              | 355      |
| possible calls (stack usage control directive) . . . . .         | 452      |
| pow (library routine). . . . .                                   | 139–140  |
| alternative implementation of . . . . .                          | 394      |
| powf (library routine) . . . . .                                 | 140      |
| powl (library routine) . . . . .                                 | 140      |
| pragma directives . . . . .                                      | 49       |

- summary . . . . . 353
- basic\_template\_matching, using . . . . . 196
- for absolute located data . . . . . 223
- implementation-defined behavior for C++ . . . . . 512
- list of all recognized . . . . . 531
- list of all recognized (C89) . . . . . 549
- pack . . . . . 331, 371
- preconfig (linker option) . . . . . 313
- predefined symbols
  - overview . . . . . 49
  - summary . . . . . 386
- predef\_macro (compiler option) . . . . . 281
- preferred\_typedef (pragma directive) . . . . . 533
- preinclude (compiler option) . . . . . 282
- .preinit\_array (section) . . . . . 445
- preprocess (compiler option) . . . . . 282
- preprocessor
  - output . . . . . 282
- preprocessor directives
  - comments at the end of . . . . . 188
  - implementation-defined behavior in C . . . . . 530
  - implementation-defined behavior in C89 . . . . . 548
  - #pragma . . . . . 353
  - #pragma (implementation-defined behavior for C++) . . . . . 512
- preprocessor extensions
  - #warning message . . . . . 392
- preprocessor symbols . . . . . 386
- defining . . . . . 259, 297
- preserved registers . . . . . 168
- \_\_PRETTY\_FUNCTION\_\_ (predefined symbol) . . . . . 389
- primitives, for special functions . . . . . 79
- print formatter, selecting . . . . . 137
- printf (library function) . . . . . 136
  - choosing formatter . . . . . 136
  - implementation-defined behavior in C . . . . . 536
  - implementation-defined behavior in C89 . . . . . 552
- \_\_printf\_args (pragma directive) . . . . . 372
- printf\_multibytes (linker option) . . . . . 313

- printing characters
  - implementation-defined behavior in C . . . . . 540
- processor cores, supported . . . . . 62
- processor operations
  - accessing . . . . . 159
  - low-level . . . . . 184, 381
- program entry label . . . . . 141
- program termination,
  - implementation-defined behavior in C . . . . . 524
- programming hints . . . . . 231
- \_\_program\_start (label) . . . . . 141
- projects
  - basic settings for . . . . . 61
  - setting up for a library . . . . . 130
- prototypes, enforcing . . . . . 284
- ptrdiff\_t (integer type) . . . . . 330
  - implementation-defined behavior for C++ . . . . . 509, 513
- PUBLIC (assembler directive) . . . . . 283
- publication date, of this guide . . . . . 2
- public\_equ (compiler option) . . . . . 282
- public\_equ (pragma directive) . . . . . 372
- push\_macro (pragma directive) . . . . . 533
- putenv (library function), absent from DLIB . . . . . 148
- putw, in stdio.h . . . . . 403

## Q

- QCCRL78 (environment variable) . . . . . 241
- qualifiers
  - const and volatile . . . . . 332
  - implementation-defined behavior . . . . . 530
  - implementation-defined behavior in C89 . . . . . 548
- ? (in reserved identifiers) . . . . . 246
- queue (library header file) . . . . . 398
- quick\_exit (library function) . . . . . 144

## R

- r (compiler option) . . . . . 260

|                                                             |         |                                                         |     |
|-------------------------------------------------------------|---------|---------------------------------------------------------|-----|
| -r (iarchive option) . . . . .                              | 492     | implementation-defined behavior in C89 . . . . .        | 547 |
| RAM                                                         |         | in assembler-level routines . . . . .                   | 166 |
| example of declaring region . . . . .                       | 92      | preserved . . . . .                                     | 168 |
| initializers copied from ROM . . . . .                      | 59      | scratch . . . . .                                       | 168 |
| running code from . . . . .                                 | 113     | .rel (ELF section) . . . . .                            | 439 |
| saving memory . . . . .                                     | 232     | .rela (ELF section) . . . . .                           | 439 |
| --ram_reserve_ranges (ismexport option) . . . . .           | 489     | --relaxed_fp (compiler option) . . . . .                | 283 |
| random (library header file) . . . . .                      | 398     | relocation errors, resolving . . . . .                  | 116 |
| random_shuffle,                                             |         | remark (diagnostic message) . . . . .                   | 247 |
| implementation-defined behavior for C++ . . . . .           | 517     | classifying for compiler . . . . .                      | 263 |
| --range (ielfdump option) . . . . .                         | 490     | classifying for linker . . . . .                        | 299 |
| range errors . . . . .                                      | 115     | enabling in compiler . . . . .                          | 284 |
| ratio (library header file) . . . . .                       | 398     | enabling in linker . . . . .                            | 314 |
| --raw (ielfdump option) . . . . .                           | 490     | --remarks (compiler option) . . . . .                   | 284 |
| read formatter, selecting . . . . .                         | 138     | --remarks (linker option) . . . . .                     | 314 |
| reading guidelines . . . . .                                | 35      | remove (library function)                               |     |
| reading, recommended . . . . .                              | 39      | implementation-defined behavior in C . . . . .          | 536 |
| realloc (library function) . . . . .                        | 75      | implementation-defined behavior in C89 (DLIB) . . . . . | 552 |
| implementation-defined behavior in C89 . . . . .            | 552     | --remove_file_path (iobjmanip option) . . . . .         | 491 |
| <i>See also</i> heap                                        |         | --remove_section (iobjmanip option) . . . . .           | 491 |
| recursive functions                                         |         | remquo, magnitude of . . . . .                          | 534 |
| avoiding . . . . .                                          | 232     | rename (ismexport directive) . . . . .                  | 470 |
| implementation-defined behavior for C++ . . . . .           | 513     | rename (library function)                               |     |
| storing data on stack . . . . .                             | 74      | implementation-defined behavior in C . . . . .          | 536 |
| --redirect (linker option) . . . . .                        | 313     | implementation-defined behavior in C89 (DLIB) . . . . . | 552 |
| reentrancy (DLIB) . . . . .                                 | 394     | --rename_section (iobjmanip option) . . . . .           | 492 |
| reference information, typographic convention . . . . .     | 40      | --rename_symbol (iobjmanip option) . . . . .            | 492 |
| regex_constants::error_type,                                |         | Renesas CC-RL compiler, linking output from . . . . .   | 115 |
| implementation-defined behavior for C++ . . . . .           | 518     | --replace (iarchive option) . . . . .                   | 492 |
| region expression (in linker configuration file) . . . . .  | 412     | __iar_ReportAssert (library function) . . . . .         | 149 |
| region literal (in linker configuration file) . . . . .     | 411     | required (pragma directive) . . . . .                   | 373 |
| register banks, switching . . . . .                         | 355     | --require_prototypes (compiler option) . . . . .        | 284 |
| register keyword, implementation-defined behavior . . . . . | 529     | reserved identifiers . . . . .                          | 246 |
| register parameters . . . . .                               | 168–169 | --reserve_ranges (ismexport option) . . . . .           | 493 |
| register variable work area. <i>See</i> workseg area        |         | restrict keyword, enabling . . . . .                    | 267 |
| registered trademarks . . . . .                             | 2       | return addresses . . . . .                              | 171 |
| registers                                                   |         | return values, from functions . . . . .                 | 170 |
| assigning to parameters . . . . .                           | 169     | RL78                                                    |     |
| callee-save, stored on stack . . . . .                      | 73      | features, support for . . . . .                         | 51  |
| for function returns . . . . .                              | 170     |                                                         |     |



supported devices . . . . . 47

RL78 0 core (legacy name) . . . . . 47

RL78 1 core (legacy name) . . . . . 47

RL78 2 core (legacy name) . . . . . 47

\_\_rollb (intrinsic function) . . . . . 383

\_\_rollw (intrinsic function) . . . . . 384

ROM to RAM, copying . . . . . 113

\_\_root (extended keyword) . . . . . 348

\_\_rorlb (intrinsic function) . . . . . 384

\_\_rorlw (intrinsic function) . . . . . 384

routines, time-critical . . . . . 159, 184, 381

\_\_ro\_placement (extended keyword) . . . . . 348

rtmodel (assembler directive) . . . . . 119

rtmodel (pragma directive) . . . . . 373

\_\_RTTI\_\_ (predefined symbol) . . . . . 389

runtime environment

    DLIB . . . . . 121

    setting up (DLIB) . . . . . 127

runtime libraries (DLIB)

    introduction . . . . . 393

    customizing system startup code . . . . . 144

    filename syntax . . . . . 134

    overriding modules in . . . . . 128

    using prebuilt . . . . . 132

runtime model attributes . . . . . 117

runtime model definitions . . . . . 374

## S

-s (ielfdump option) . . . . . 494

\_\_saddr (extended keyword) . . . . . 349

saddr (memory type) . . . . . 66

.sbss (ELF section) . . . . . 445

.sbss.noinit (ELF section) . . . . . 445

scanf (library function)

    choosing formatter (DLIB) . . . . . 137

    implementation-defined behavior in C . . . . . 536

    implementation-defined behavior in C89 (DLIB) . . . . . 552

\_\_scanf\_args (pragma directive) . . . . . 374

--scanf\_multibytes (linker option) . . . . . 314

scheduling (compiler transformation) . . . . . 231

    disabling . . . . . 276

scoped\_allocator (library header file) . . . . . 399

scratch registers . . . . . 168

.sdata (ELF section) . . . . . 445

.sdata\_init (ELF section) . . . . . 446

--search (linker option) . . . . . 314

search directory, for linker configuration files

    (--config\_search) . . . . . 295

search path to library files (--search) . . . . . 314

search path to object files (--search) . . . . . 314

--section (ielfdump option) . . . . . 494

sections . . . . . 437

    summary . . . . . 437

    allocation of . . . . . 91

    checking type at link-time . . . . . 407

    declaring (#pragma section) . . . . . 374

    introduction . . . . . 88

    specifying (--code\_section) . . . . . 258

\_\_section\_begin (extended operator) . . . . . 186

\_\_section\_end (extended operator) . . . . . 186

\_\_section\_size (extended operator) . . . . . 186

section-selectors (in linker configuration file) . . . . . 427

.security\_id (section) . . . . . 446

--segment (ielfdump option) . . . . . 494

segment (pragma directive) . . . . . 374

--self\_reloc (ielftool option) . . . . . 495

semaphores

    C example . . . . . 80

    C++ example . . . . . 82

    operations on . . . . . 343

separate\_init\_routine (pragma directive) . . . . . 533

separating section parts, preventing . . . . . 115

set (library header file) . . . . . 399

setjmp.h (library header file) . . . . . 396

setlocale (library function) . . . . . 155

settings, basic for project configuration . . . . . 61

set\_generate\_entries\_without\_bounds (pragma directive) . 533

\_\_set\_interrupt\_level (intrinsic function) . . . . . 384

|                                                            |          |                                                                    |               |
|------------------------------------------------------------|----------|--------------------------------------------------------------------|---------------|
| __set_interrupt_state (intrinsic function) . . . . .       | 384      | --simple-ne (ielftool option) . . . . .                            | 496           |
| severity level, of diagnostic messages . . . . .           | 247      | sin (library function) . . . . .                                   | 394           |
| specifying . . . . .                                       | 248      | sin (library routine) . . . . .                                    | 139–140       |
| SFR . . . . .                                              |          | sinf (library routine) . . . . .                                   | 140           |
| accessing special function registers . . . . .             | 234      | sinl (library routine) . . . . .                                   | 140           |
| declaring extern special function registers . . . . .      | 224      | 64-bits (floating-point format) . . . . .                          | 327           |
| __sfr (extended keyword) . . . . .                         | 349      | size (in stack usage control file) . . . . .                       | 455           |
| SFR (memory type) . . . . .                                | 67       | sizeof, implementation-defined behavior for C++ . . . . .          | 509           |
| shared object . . . . .                                    | 243, 307 | size_t (integer type) . . . . .                                    | 329           |
| shared_mutex (library header file) . . . . .               | 399      | implementation-defined behavior for C++ . . . . .                  | 513           |
| shared_ptr constructor, . . . . .                          |          | skeleton code, creating for assembler language interface . . . . . | 163           |
| implementation-defined behavior for C++ . . . . .          | 514      | slist (library header file) . . . . .                              | 399           |
| short address work area. <i>See</i> workseg area . . . . . |          | small function inlining (linker optimization) . . . . .            | 120           |
| short (data type) . . . . .                                | 323      | smallest, packing algorithm for initializers . . . . .             | 419           |
| show (ismlexport directive) . . . . .                      | 470      | --source (ielfdump option) . . . . .                               | 496           |
| --show_entry_as (ismlexport option) . . . . .              | 495      | source files, list all referred . . . . .                          | 270           |
| show-root (ismlexport directive) . . . . .                 | 471      | --source_encoding (compiler option) . . . . .                      | 285           |
| show-weak (ismlexport directive) . . . . .                 | 471      | space characters, implementation-defined behavior in C . . . . .   | 535           |
| .shstrtab (ELF section) . . . . .                          | 439      | special function registers (SFR) . . . . .                         | 234           |
| signal (library function) . . . . .                        |          | special function types . . . . .                                   | 79            |
| implementation-defined behavior in C . . . . .             | 534      | sprintf (library function) . . . . .                               | 136           |
| implementation-defined behavior in C89 . . . . .           | 551      | choosing formatter . . . . .                                       | 136           |
| signals, implementation-defined behavior in C . . . . .    | 524      | --src (ielftool option) . . . . .                                  | 497           |
| at system startup . . . . .                                | 525      | --src-len (ielftool option) . . . . .                              | 497           |
| signal.h (library header file) . . . . .                   | 396      | --src-s3only (ielftool option) . . . . .                           | 497           |
| signed char (data type) . . . . .                          | 323      | sscanf (library function) . . . . .                                |               |
| specifying . . . . .                                       | 257      | choosing formatter (DLIB) . . . . .                                | 137           |
| signed int (data type) . . . . .                           | 323      | sstream (library header file) . . . . .                            | 399           |
| signed long long (data type) . . . . .                     | 323      | stack . . . . .                                                    | 73            |
| signed long (data type) . . . . .                          | 323      | advantages and problems using . . . . .                            | 74            |
| signed short (data type) . . . . .                         | 323      | block for holding . . . . .                                        | 441           |
| signed values, avoiding . . . . .                          | 219      | cleaning . . . . .                                                 | 171, 256, 350 |
| --silent (compiler option) . . . . .                       | 284      | contents of . . . . .                                              | 73            |
| --silent (iarchive option) . . . . .                       | 495      | layout . . . . .                                                   | 170           |
| --silent (ielftool option) . . . . .                       | 495      | saving space . . . . .                                             | 232           |
| --silent (linker option) . . . . .                         | 315      | setting up size for . . . . .                                      | 109           |
| silent operation . . . . .                                 |          | size . . . . .                                                     | 206           |
| specifying in compiler . . . . .                           | 284      | stack buffer overflow . . . . .                                    | 85            |
| specifying in linker . . . . .                             | 315      | stack buffer overrun . . . . .                                     | 85            |
| --simple (ielftool option) . . . . .                       | 496      |                                                                    |               |

- stack canaries . . . . . 85
- stack parameters . . . . . 168, 170
- stack pointer . . . . . 73
- stack pointer register, considerations . . . . . 168
- stack protection . . . . . 85
- stack smashing . . . . . 85
- stack (library header file) . . . . . 399
- stack\_protect (pragma directive) . . . . . 375
- sack\_protection (compiler option) . . . . . 285
- stack\_usage\_control (linker option) . . . . . 315
- stack-size (in stack usage control file) . . . . . 454
- Standard C . . . . . 267
  - library compliance with . . . . . 393
  - specifying strict usage . . . . . 285
- Standard C++
  - implementation quantities . . . . . 519
  - implementation-defined behavior . . . . . 503
- standard error
  - redirecting in compiler . . . . . 280
  - redirecting in linker . . . . . 311
  - See also diagnostic messages . . . . . 243
- standard library
  - functions, implementation-defined behavior for C++ . . . . . 513
- standard output
  - specifying in compiler . . . . . 280
  - specifying in linker . . . . . 311
- start up system. *See* system startup
- startup code
  - cstartup . . . . . 144
- statements, implementation-defined behavior in C89 . . . . . 548
- static analysis
  - documentation for . . . . . 38
- static clustering (compiler transformation) . . . . . 230
- static variables . . . . . 65
  - taking the address of . . . . . 232
- status flags for floating-point . . . . . 402
- stdalign.h (library header file) . . . . . 396
- stdarg.h (library header file) . . . . . 396
- stdatomic.h (library header file) . . . . . 396
- stdbool.h (library header file) . . . . . 323, 396
- \_\_STDC\_\_ (predefined symbol) . . . . . 389
  - implementation-defined behavior for C++ . . . . . 512
- STDC CX\_LIMITED\_RANGE (pragma directive) . . . . . 375
- STDC FENV\_ACCESS (pragma directive) . . . . . 376
- STDC FP\_CONTRACT (pragma directive) . . . . . 376
- \_\_STDC\_LIB\_EXT1\_\_ (predefined symbol) . . . . . 389
- \_\_STDC\_NO\_ATOMICS\_\_ (preprocessor symbol) . . . . . 390
- \_\_STDC\_NO\_THREADS\_\_ (preprocessor symbol) . . . . . 390
- \_\_STDC\_NO\_VLA\_\_ (preprocessor symbol) . . . . . 390
- \_\_STDC\_UTF16\_\_ (preprocessor symbol) . . . . . 390
- \_\_STDC\_UTF32\_\_ (preprocessor symbol) . . . . . 390
- \_\_STDC\_VERSION\_\_ (predefined symbol) . . . . . 390
  - implementation-defined behavior for C++ . . . . . 512
- \_\_STDC\_WANT\_LIB\_EXT1\_\_ (preprocessor symbol) . . . . . 392
- stddef.h (library header file) . . . . . 396
- stderr . . . . . 127, 280, 311
- stdexcept (library header file) . . . . . 399
- stdin . . . . . 127
  - implementation-defined behavior in C89 (DLIB) . . . . . 551
- stdint.h (library header file) . . . . . 396, 400
- stdio.h (library header file) . . . . . 396
- stdio.h, additional C functionality . . . . . 402
- stdlib.h (library header file) . . . . . 396
- stdnoreturn.h (library header file) . . . . . 397
- stdout . . . . . 127, 280, 311
  - implementation-defined behavior in C . . . . . 534
  - implementation-defined behavior in C89 (DLIB) . . . . . 551
- std::terminate, implementation-defined behavior for C++ . . . . . 511
- std::unexpected,
  - implementation-defined behavior for C++ . . . . . 511
- Steele, Guy L. . . . . 39
- steering file, input to isymexport . . . . . 469
- \_\_stop (intrinsic function) . . . . . 384
- strcasecmp, in string.h . . . . . 403
- strcoll (function) . . . . . 404
- strdup, in string.h . . . . . 403
- streambuf (library header file) . . . . . 399
- streamoff, implementation-defined behavior for C++ . . . . . 515
- streampos, implementation-defined behavior for C++ . . . . . 515

streams  
 implementation-defined behavior in C . . . . . 524

strerror (library function)  
 implementation-defined behavior in C . . . . . 541

strerror (library function),  
 implementation-defined behavior in C89 (DLIB) . . . . . 553

--strict (compiler option) . . . . . 285

string literals, compiler padding . . . . . 322

string literals, implementation-defined behavior for C++ . 506

string (library header file) . . . . . 399

string.h (library header file) . . . . . 397

string.h, additional C functionality . . . . . 403

--strip (ielftool option) . . . . . 498

--strip (objmanip option) . . . . . 498

--strip (linker option) . . . . . 315

strncasecmp, in string.h . . . . . 403

strlen, in string.h . . . . . 403

strstream (library header file) . . . . . 399

.strtab (ELF section) . . . . . 439

strtod (library function), configuring support for . . . . . 156

structure types  
 alignment . . . . . 330–331  
 layout of . . . . . 330  
 packed . . . . . 331

structures  
 accessing using a pointer . . . . . 175  
 aligning . . . . . 371  
 anonymous . . . . . 220  
 implementation-defined behavior in C . . . . . 529  
 implementation-defined behavior in C89 . . . . . 547  
 packing and unpacking . . . . . 220  
 placing in memory type . . . . . 70

strxfrm (function) . . . . . 404

subnormal numbers . . . . . 328

\$Sub\$\$ pattern . . . . . 218

\$Super\$\$ pattern . . . . . 218

support, technical . . . . . 248

Sutter, Herb . . . . . 39

.switch (section) . . . . . 446

.switchf (section) . . . . . 446

symbol names, prefixed by extra underscore . . . . . 294, 449

symbols  
 directing from one to another . . . . . 313  
 including in output . . . . . 373  
 local, removing from ELF image . . . . . 310  
 overview of predefined . . . . . 49  
 patching using \$Super\$\$ and \$Sub\$\$ . . . . . 218  
 preprocessor, defining . . . . . 259, 297

--symbols (iarchive option) . . . . . 498

.symtab (ELF section) . . . . . 439

syntax  
 command line options . . . . . 249  
 extended keywords . . . . . 68, 336–337, 339  
 invoking compiler and linker . . . . . 239

system function, implementation-defined behavior in C . . 537

system function,  
 implementation-defined behavior in C . . . . . 525

system startup  
 customizing . . . . . 144  
 DLIB . . . . . 141  
 implementation-defined behavior for C++ . . . . . 506  
 initialization phase . . . . . 57

system termination  
 C-SPY interface to . . . . . 144  
 DLIB . . . . . 143  
 implementation-defined behavior for C++ . . . . . 506

system (library function)  
 implementation-defined behavior in C89 (DLIB) . . . . 553

system\_error (library header file) . . . . . 399

system\_include (pragma directive) . . . . . 533, 550

--system\_include\_dir (compiler option) . . . . . 286

## T

-t (iarchive option) . . . . . 499

tan (library function) . . . . . 394

tan (library routine) . . . . . 139–140

tanf (library routine) . . . . . 140

tanl (library routine) . . . . . 140

- technical support, IAR Systems . . . . . 248
- template support
  - in C++ . . . . . 195
- Terminal I/O window
  - not supported when . . . . . 128–129
- termination of system. *See* system termination
- termination status, implementation-defined behavior in C 537
- terminology. . . . . 39
- .text (ELF section) . . . . . 446
- text encodings . . . . . 244
- .textf (ELF section) . . . . . 447
- text\_out (iarchive option) . . . . . 498
- text\_out (ielfdump option) . . . . . 498
- text\_out (iobjmanip option) . . . . . 498
- text\_out (isymexport option) . . . . . 498
- text\_out (linker option) . . . . . 316
- text\_out (compiler option) . . . . . 286
- .text\_unit64kp (ELF section) . . . . . 438
- tgmath.h (library header file) . . . . . 397
- 32-bits (floating-point format) . . . . . 327
- this (pointer) . . . . . 165
  - class memory . . . . . 193
  - referring to a class object . . . . . 193
- thread (library header file) . . . . . 399
- threaded environment . . . . . 157
- threaded\_lib (linker option) . . . . . 316
- threads, number of
  - (implementation-defined behavior for C++) . . . . . 504
- threads.h (library header file) . . . . . 397
- \_\_TIME\_\_ (predefined symbol) . . . . . 391
  - implementation-defined behavior for C++ . . . . . 512
- time zone (library function)
  - implementation-defined behavior in C89 . . . . . 553
  - time zone (library function), implementation-defined behavior in C . . . . . 537
- \_\_TIMESTAMP\_\_ (predefined symbol) . . . . . 391
- timezone\_lib (linker option) . . . . . 316
- time\_get::do\_get\_date,
  - implementation-defined behavior for C++ . . . . . 516
- time\_get::do\_get\_year,
  - implementation-defined behavior for C++ . . . . . 516
- time\_put::do\_put,
  - implementation-defined behavior for C++ . . . . . 516
- time\_t value to time\_point object
  - conversion, implementation-defined behavior for C++ . . . 515
- time-critical routines . . . . . 159, 184, 381
- time.h (library header file) . . . . . 397
  - additional C functionality . . . . . 403
- time32 (library function), configuring support for . . . . . 128
- time64 (library function), configuring support for . . . . . 128
- tips, programming . . . . . 231
- titxt (ielftool option) . . . . . 499
- toc (iarchive option) . . . . . 499
- tokens, attribute-
  - scoped (implementation-defined behavior for C++) . . . . 510
- tools icon, in this guide . . . . . 40
- tolower (function) . . . . . 404
- toupper (function) . . . . . 404
- trademarks . . . . . 2
- trailing comma . . . . . 203
- transformations, compiler . . . . . 226
- translation
  - implementation-defined behavior . . . . . 523
  - implementation-defined behavior for C++ . . . . . 503, 505
  - implementation-defined behavior in C89 . . . . . 543
- trap vectors, specifying with pragma directive . . . . . 378
- tuple (library header file) . . . . . 399
- type attributes . . . . . 335
  - specifying . . . . . 377
- type definitions, used for specifying memory storage . . . . 69
- type qualifiers
  - const and volatile . . . . . 332
  - implementation-defined behavior . . . . . 530
  - implementation-defined behavior in C89 . . . . . 548
- typedefs
  - excluding from diagnostics . . . . . 278
  - repeated . . . . . 188
- typeid, derived
- type for (implementation-defined behavior for C++) . . . . 509

typeindex (library header file) . . . . . 399  
 typeid (library header file) . . . . . 399  
 types, trivially  
   copyable (implementation-defined behavior for C++) . . . 507  
   typetraits (library header file) . . . . . 399  
   type\_attribute (pragma directive) . . . . . 377  
   type\_info::name,  
   implementation-defined behavior for C++ . . . . . 514  
 type-based alias analysis (compiler transformation) . . . . 230  
   disabling . . . . . 277  
 typographic conventions . . . . . 40

## U

uchar.h (library header file) . . . . . 397  
 uintptr\_t (integer type) . . . . . 330  
 underflow errors, implementation-defined behavior in C . 534  
 underflow range errors,  
 implementation-defined behavior in C89 . . . . . 550  
 underscore  
   double in reserved identifiers . . . . . 246  
   extra before assembler labels . . . . . 294  
   followed by uppercase letter (reserved identifier) . . . 246  
 underscore, extra before assembler labels . . . . . 449  
 \_\_ungetchar, in stdio.h . . . . . 403  
 Unicode . . . . . 245  
 uniform attribute syntax . . . . . 68, 336  
 --uniform\_attribute\_syntax (compiler option) . . . . . 287  
 unions  
   anonymous . . . . . 220  
   implementation-defined behavior in C . . . . . 529  
   implementation-defined behavior in C89 . . . . . 547  
 universal character names, implementation-defined  
 behavior in C . . . . . 531  
 universal character  
 names, implementation-defined behavior for C++ . . . . . 506  
 unordered\_map (library header file) . . . . . 399  
   implementation-defined behavior for C++ . . . . . 517  
 unordered\_multimap,  
 implementation-defined behavior for C++ . . . . . 517

unordered\_multiset,  
 implementation-defined behavior for C++ . . . . . 517  
 unordered\_set (library header file) . . . . . 399  
   implementation-defined behavior for C++ . . . . . 517  
 unroll (pragma directive) . . . . . 377  
 unsigned char (data type) . . . . . 323  
   changing to signed char . . . . . 257  
 unsigned int (data type) . . . . . 323  
 unsigned long long (data type) . . . . . 323  
 unsigned long (data type) . . . . . 323  
 unsigned short (data type) . . . . . 323  
 unsigned to signed  
 conversion, implementation-defined behavior for C++ . . . 508  
 use init format (linker directive) . . . . . 425  
 use init table (linker directive) . . . . . 425  
 uses\_aspect (pragma directive) . . . . . 533  
 --use\_cplusplus\_inline (compiler option) . . . . . 287  
 --use\_full\_std\_template\_names (ielfdump option) . . . . . 500  
 --use\_full\_std\_template\_names (linker option) . . . . . 317  
 --use\_paths\_as\_written (compiler option) . . . . . 287  
 --use\_unix\_directory\_separators (compiler option) . . . . . 288  
 UTF-16 . . . . . 245  
 UTF-8 . . . . . 245  
 --utf8\_text\_in (compiler option) . . . . . 288  
 --utf8\_text\_in (iarchive option) . . . . . 500  
 --utf8\_text\_in (ielfdump option) . . . . . 500  
 --utf8\_text\_in (iobjmanip option) . . . . . 500  
 --utf8\_text\_in (isymexport option) . . . . . 500  
 --utf8\_text\_in (linker option) . . . . . 317  
 utilities (ELF) . . . . . 457  
 utility (library header file) . . . . . 399  
 u16streampos, implementation-defined behavior for C++ 515  
 u32streampos, implementation-defined behavior for C++ 515

## V

-V (iarchive option) . . . . . 500  
 valarray (library header file) . . . . . 399

- variables
    - auto . . . . . 73
    - defined inside a function . . . . . 73
    - global
      - accessing . . . . . 175
      - placement in memory . . . . . 65
    - hints for choosing . . . . . 232
    - local. *See* auto variables
    - non-initialized . . . . . 236
    - placing at absolute addresses . . . . . 224
    - placing in named sections . . . . . 224
    - static
      - placement in memory . . . . . 65
      - taking the address of . . . . . 232
  - .vector (ELF section) . . . . . 438
  - vector (library header file) . . . . . 399
  - vector (pragma directive) . . . . . 80, 378
  - verbose (iarchive option) . . . . . 500
  - verbose (ielftool option) . . . . . 500
  - version
    - compiler subversion number . . . . . 390
    - identifying C standard in use (`__STDC_VERSION__`) 390
    - of compiler (`__VER__`) . . . . . 391
    - of this guide . . . . . 2
  - version (linker option) . . . . . 317
  - version (compiler option) . . . . . 288
  - version (utilities option) . . . . . 501
  - vfe (linker option) . . . . . 318
  - virtual function elimination (linker optimization) . . . . . 119
  - vla (compiler option) . . . . . 289
  - void, pointers to . . . . . 188
  - volatile
    - and const, declaring objects . . . . . 333
    - declaring objects . . . . . 332
    - protecting simultaneously accesses variables. . . . . 234
    - rules for access. . . . . 333
  - volatile-qualified
  - type, implementation-defined behavior for C++ . . . . . 510
  - vtoc (iarchive option) . . . . . 501
  - V1 calling convention . . . . . 166
  - \_\_v1\_call (extended keyword) . . . . . 350
  - V2 calling convention . . . . . 166
  - \_\_v2\_call (extended keyword) . . . . . 350
- ## W
- #warning message (preprocessor extension) . . . . . 392
  - warnings . . . . . 247
    - classifying in compiler . . . . . 263
    - classifying in linker . . . . . 300
    - disabling in compiler . . . . . 279
    - disabling in linker . . . . . 311
    - exit code in compiler . . . . . 289
    - exit code in linker . . . . . 318
  - warnings icon, in this guide . . . . . 40
  - warnings (pragma directive) . . . . . 533, 550
  - warnings\_affect\_exit\_code (compiler option) . . . . . 243, 289
  - warnings\_affect\_exit\_code (linker option) . . . . . 318
  - warnings\_are\_errors (compiler option) . . . . . 289
  - warnings\_are\_errors (linker option) . . . . . 318
  - warn\_about\_c\_style\_casts (compiler option) . . . . . 289
  - wchar\_t (data type) . . . . . 324
    - implementation-defined behavior in C . . . . . 527
  - wchar.h (library header file) . . . . . 397, 400
  - wctype.h (library header file) . . . . . 397
  - \_\_weak (extended keyword) . . . . . 351
  - weak (pragma directive) . . . . . 378
  - web sites, recommended . . . . . 39
  - white-space characters, implementation-defined behavior 523
  - whole\_archive (linker option) . . . . . 319
  - wide-character
    - literals, implementation-defined behavior for C++ . . . . . 505
  - workseg area . . . . . 74
  - workseg\_area (compiler option) . . . . . 290
  - \_\_write\_array, in `stdio.h` . . . . . 403
  - \_\_write\_buffered (DLIB library function) . . . . . 126
  - .wrkseg (ELF section) . . . . . 447
  - wstreampos, implementation-defined behavior for C++ . . 515

# X

-x (iarchive option) . . . . . 482

# Z

zeros, packing algorithm for initializers . . . . . 419

# Symbols

|                                                         |          |
|---------------------------------------------------------|----------|
| __Exit (library function) . . . . .                     | 143      |
| __exit (library function) . . . . .                     | 143      |
| __ALIGNOF__ (operator) . . . . .                        | 186      |
| __asm (language extension) . . . . .                    | 161      |
| __BASE_FILE__ (predefined symbol) . . . . .             | 386      |
| __break (intrinsic function) . . . . .                  | 382      |
| __BUILD_NUMBER__ (predefined symbol) . . . . .          | 386      |
| __CALLING_CONVENTION__ (predefined symbol) . . . . .    | 386      |
| __callt runtime library calls, generating . . . . .     | 269      |
| __callt (extended keyword) . . . . .                    | 340      |
| __CODE_MODEL__ (predefined symbol) . . . . .            | 386      |
| __CORE__ (predefined symbol) . . . . .                  | 386      |
| __COUNTER__ (predefined symbol) . . . . .               | 387      |
| __cplusplus (predefined symbol) . . . . .               | 387      |
| __DATA_MODEL__ (predefined symbol) . . . . .            | 387      |
| __DATE__ (predefined symbol) . . . . .                  | 387      |
| implementation-defined behavior for C++ . . . . .       | 512      |
| __disable_interrupt (intrinsic function) . . . . .      | 382      |
| __DLIB_FILE_DESCRIPTOR (configuration symbol) . . . . . | 154      |
| __enable_interrupt (intrinsic function) . . . . .       | 382      |
| __EXCEPTIONS__ (predefined symbol) . . . . .            | 387      |
| __exit (library function) . . . . .                     | 143      |
| __far (extended keyword) . . . . .                      | 329, 341 |
| __far_func (extended keyword) . . . . .                 | 341      |
| __far_func (function pointer) . . . . .                 | 328      |
| __FAR_RUNTIME_ATTRIBUTE__ (predefined symbol) . . . . . | 387      |
| __FILE__ (predefined symbol) . . . . .                  | 388      |
| __FUNCTION__ (predefined symbol) . . . . .              | 388      |

|                                                      |          |
|------------------------------------------------------|----------|
| __func__ (predefined symbol) . . . . .               | 388      |
| implementation-defined behavior for C++ . . . . .    | 510      |
| __gets, in stdio.h . . . . .                         | 402      |
| __get_interrupt_level (intrinsic function) . . . . . | 382      |
| __get_interrupt_state (intrinsic function) . . . . . | 382      |
| __halt (intrinsic function) . . . . .                | 383      |
| __huge (extended keyword) . . . . .                  | 329, 342 |
| __iar_cos_accurate (library routine) . . . . .       | 140      |
| __iar_cos_accuratef (library routine) . . . . .      | 140      |
| __iar_cos_accuratel (library routine) . . . . .      | 140      |
| __iar_cos_small (library routine) . . . . .          | 139      |
| __iar_cos_smallf (library routine) . . . . .         | 140      |
| __iar_cos_smallll (library routine) . . . . .        | 140      |
| __iar_exp_small (library routine) . . . . .          | 139      |
| __iar_exp_smallf (library routine) . . . . .         | 140      |
| __iar_exp_smallll (library routine) . . . . .        | 140      |
| __iar_log_small (library routine) . . . . .          | 139      |
| __iar_log_smallf (library routine) . . . . .         | 140      |
| __iar_log_smallll (library routine) . . . . .        | 140      |
| __iar_log10_small (library routine) . . . . .        | 139      |
| __iar_log10_smallf (library routine) . . . . .       | 140      |
| __iar_log10_smallll (library routine) . . . . .      | 140      |
| __iar_maximum_atexit_calls . . . . .                 | 110      |
| __iar_pow_accurate (library routine) . . . . .       | 140      |
| __iar_pow_accuratef (library routine) . . . . .      | 140      |
| __iar_pow_accuratel (library routine) . . . . .      | 140      |
| __iar_pow_small (library routine) . . . . .          | 139      |
| __iar_pow_smallf (library routine) . . . . .         | 140      |
| __iar_pow_smallll (library routine) . . . . .        | 140      |
| __iar_program_start (label) . . . . .                | 141      |
| __iar_ReportAssert (library function) . . . . .      | 149      |
| __iar_sin_accurate (library routine) . . . . .       | 140      |
| __iar_sin_accuratef (library routine) . . . . .      | 140      |
| __iar_sin_accuratel (library routine) . . . . .      | 140      |
| __iar_sin_small (library routine) . . . . .          | 139      |
| __iar_sin_smallf (library routine) . . . . .         | 140      |
| __iar_sin_smallll (library routine) . . . . .        | 140      |
| __IAR_SYSTEMS_ICC__ (predefined symbol) . . . . .    | 388      |
| __iar_tan_accurate (library routine) . . . . .       | 140      |



- `__iar_tan_accuratef` (library routine) . . . . . 140
- `__iar_tan_accuratel` (library routine) . . . . . 140
- `__iar_tan_small` (library routine) . . . . . 139
- `__iar_tan_smallf` (library routine) . . . . . 140
- `__iar_tan_smalll` (library routine) . . . . . 140
- `__iar_tls.$DATA` (ELF section) . . . . . 444
- `__ICCRL78__` (predefined symbol) . . . . . 388
- `__interrupt` (extended keyword) . . . . . 80, 342
  - using in pragma directives . . . . . 378
- `__intrinsic` (extended keyword) . . . . . 343
- `__LINE__` (predefined symbol) . . . . . 389
- `__low_level_init` . . . . . 141
  - initialization phase . . . . . 57
  - customizing . . . . . 144
- `__mach` (intrinsic function) . . . . . 383
- `__machu` (intrinsic function) . . . . . 383
- `__memory_of`
  - operator . . . . . 194
- `__monitor` (extended keyword) . . . . . 343
- `__near` (extended keyword) . . . . . 329, 343
- `__near_func` (extended keyword) . . . . . 344
- `__near_func` (function pointer) . . . . . 328
- `__near_size_t` . . . . . 199
- `__noreturn` (extended keyword) . . . . . 346
- `__no_alloc` (extended keyword) . . . . . 344
- `__no_alloc_str` (operator) . . . . . 344–345
- `__no_alloc_str16` (operator) . . . . . 344–345
- `__no_alloc16` (extended keyword) . . . . . 344
- `__no_bit_access` (extended keyword) . . . . . 345
- `__no_init` (extended keyword) . . . . . 236, 346
- `__no_operation` (intrinsic function) . . . . . 383
- `__no_save` (extended keyword) . . . . . 346
- `__packed` (extended keyword) . . . . . 347
- `__PRETTY_FUNCTION__` (predefined symbol) . . . . . 389
- `__printf_args` (pragma directive) . . . . . 372
- `__program_start` (label) . . . . . 141
- `__rollb` (intrinsic function) . . . . . 383
- `__rollw` (intrinsic function) . . . . . 384
- `__root` (extended keyword) . . . . . 348
- `__ror1b` (intrinsic function) . . . . . 384
- `__ror1w` (intrinsic function) . . . . . 384
- `__ro_placement` (extended keyword) . . . . . 348
- `__RTTI__` (predefined symbol) . . . . . 389
- `__saddr` (extended keyword) . . . . . 349
- `__scanf_args` (pragma directive) . . . . . 374
- `__section_begin` (extended operator) . . . . . 186
- `__section_end` (extended operator) . . . . . 186
- `__section_size` (extended operator) . . . . . 186
- `__set_interrupt_level` (intrinsic function) . . . . . 384
- `__set_interrupt_state` (intrinsic function) . . . . . 384
- `__sfr` (extended keyword) . . . . . 349
- `__STDC_LIB_EXT1__` (predefined symbol) . . . . . 389
- `__STDC_NO_ATOMICS__` (preprocessor symbol) . . . . . 390
- `__STDC_NO_THREADS__` (preprocessor symbol) . . . . . 390
- `__STDC_NO_VLA__` (preprocessor symbol) . . . . . 390
- `__STDC_UTF16__` (preprocessor symbol) . . . . . 390
- `__STDC_UTF32__` (preprocessor symbol) . . . . . 390
- `__STDC_VERSION__` (predefined symbol) . . . . . 390
  - implementation-defined behavior for C++ . . . . . 512
- `__STDC_WANT_LIB_EXT1__` (preprocessor symbol) . . . . . 392
- `__STDC__` (predefined symbol) . . . . . 389
  - implementation-defined behavior for C++ . . . . . 512
- `__stop` (intrinsic function) . . . . . 384
- `__TIMESTAMP__` (predefined symbol) . . . . . 391
  - implementation-defined behavior for C++ . . . . . 512
- `__ungetchar`, in `stdio.h` . . . . . 403
- `__v1_call` (extended keyword) . . . . . 350
- `__v2_call` (extended keyword) . . . . . 350
- `__weak` (extended keyword) . . . . . 351
- `__write_array`, in `stdio.h` . . . . . 403
- `__write_buffered` (DLIB library function) . . . . . 126
- `-D` (compiler option) . . . . . 259
- `-d` (iarchive option) . . . . . 480
- `-e` (compiler option) . . . . . 267
- `-f` (compiler option) . . . . . 268
- `-f` (IAR utility option) . . . . . 482
- `-f` (linker option) . . . . . 302

|                                                       |          |                                                                         |     |
|-------------------------------------------------------|----------|-------------------------------------------------------------------------|-----|
| -g (ielfdump option) . . . . .                        | 494      | --define_symbol (linker option) . . . . .                               | 297 |
| -I (compiler option) . . . . .                        | 270      | --delete (iarchive option) . . . . .                                    | 480 |
| -l (compiler option) . . . . .                        | 270      | --dependencies (compiler option) . . . . .                              | 260 |
| for creating skeleton code . . . . .                  | 164      | --dependencies (linker option) . . . . .                                | 297 |
| -L (linker option) . . . . .                          | 314      | --deprecated_feature_warnings (compiler option) . . . . .               | 262 |
| -O (compiler option) . . . . .                        | 280      | --diagnostics_tables (compiler option) . . . . .                        | 264 |
| -o (compiler option) . . . . .                        | 281      | --diagnostics_tables (linker option) . . . . .                          | 300 |
| -o (iarchive option) . . . . .                        | 488      | --diag_error (compiler option) . . . . .                                | 262 |
| -o (ielfdump option) . . . . .                        | 488      | --diag_error (linker option) . . . . .                                  | 298 |
| -o (linker option) . . . . .                          | 312      | --diag_remark (compiler option) . . . . .                               | 263 |
| -r (compiler option) . . . . .                        | 260      | --diag_remark (linker option) . . . . .                                 | 299 |
| -r (iarchive option) . . . . .                        | 492      | --diag_suppress (compiler option) . . . . .                             | 263 |
| -s (ielfdump option) . . . . .                        | 494      | --diag_suppress (linker option) . . . . .                               | 299 |
| -t (iarchive option) . . . . .                        | 499      | --diag_warning (compiler option) . . . . .                              | 263 |
| -V (iarchive option) . . . . .                        | 500      | --diag_warning (linker option) . . . . .                                | 300 |
| -x (iarchive option) . . . . .                        | 482      | --disable_div_mod_instructions (compiler option) . . . . .              | 264 |
| --a (ielfdump option) . . . . .                       | 473      | --disasm_data (ielfdump option) . . . . .                               | 481 |
| --all (ielfdump option) . . . . .                     | 474      | --discard_unused_publics (compiler option) . . . . .                    | 264 |
| --bin (ielftool option) . . . . .                     | 474      | --dlib_config (compiler option) . . . . .                               | 265 |
| --bin-multi (ielftool option) . . . . .               | 475      | --double (compiler option) . . . . .                                    | 266 |
| --calling_convention (compiler option) . . . . .      | 256      | --do_explicit_zero_opt_in_named_sections<br>(compiler option) . . . . . | 266 |
| --call_graph (linker option) . . . . .                | 294      | --edit (isymexport option) . . . . .                                    | 481 |
| --char_is_signed (compiler option) . . . . .          | 257      | --enable_restrict (compiler option) . . . . .                           | 267 |
| --char_is_unsigned (compiler option) . . . . .        | 257      | --entry (linker option) . . . . .                                       | 301 |
| --checksum (ielftool option) . . . . .                | 475      | --entry_list_in_address_order (linker option) . . . . .                 | 301 |
| --code (ielfdump option) . . . . .                    | 480      | --error_limit (compiler option) . . . . .                               | 267 |
| --code_model (compiler option) . . . . .              | 257, 260 | --error_limit (linker option) . . . . .                                 | 301 |
| --code_section (compiler option) . . . . .            | 258      | --export_builtin_config (linker option) . . . . .                       | 302 |
| --config (linker option) . . . . .                    | 295      | --export_locals (isymexport option) . . . . .                           | 481 |
| --config_def (linker option) . . . . .                | 295      | --extract (iarchive option) . . . . .                                   | 482 |
| --config_search (linker option) . . . . .             | 295      | --f (compiler option) . . . . .                                         | 268 |
| --core (compiler option) . . . . .                    | 258      | --f (IAR utility option) . . . . .                                      | 483 |
| --cpp_init_routine (linker option) . . . . .          | 296      | --f (linker option) . . . . .                                           | 302 |
| --create (iarchive option) . . . . .                  | 480      | --fill (ielftool option) . . . . .                                      | 484 |
| --c++ (compiler option) . . . . .                     | 259      | --force_output (linker option) . . . . .                                | 303 |
| --c89 (compiler option) . . . . .                     | 256      | --front_headers (ielftool option) . . . . .                             | 484 |
| --debug (compiler option) . . . . .                   | 260      | --generate_callt_runtime_library_calls<br>(compiler option) . . . . .   | 269 |
| --debug_lib (linker option) . . . . .                 | 296      |                                                                         |     |
| --default_to_complex_ranges (linker option) . . . . . | 297      |                                                                         |     |

|                                                                     |     |                                                         |     |
|---------------------------------------------------------------------|-----|---------------------------------------------------------|-----|
| --generate_far_runtime_library_calls<br>(compiler option) . . . . . | 269 | --no_inline (linker option). . . . .                    | 309 |
| --generate_vfe_header (ismlexport option) . . . . .                 | 485 | --no_library_search (linker option) . . . . .           | 309 |
| --guard_calls (compiler option). . . . .                            | 269 | --no_locals (linker option) . . . . .                   | 310 |
| --header_context (compiler option). . . . .                         | 270 | --no_path_in_file_macros (compiler option). . . . .     | 276 |
| --ihex (ielftool option) . . . . .                                  | 485 | --no_range_reservations (linker option) . . . . .       | 310 |
| --image_input (linker option) . . . . .                             | 303 | --no_rel_section (ielfdump option) . . . . .            | 486 |
| --inline (linker option) . . . . .                                  | 304 | --no_remove (linker option) . . . . .                   | 310 |
| --keep (linker option) . . . . .                                    | 304 | --no_scheduling (compiler option) . . . . .             | 276 |
| --log (linker option) . . . . .                                     | 305 | --no_size_constraints (compiler option) . . . . .       | 276 |
| --log_file (linker option) . . . . .                                | 306 | --no_static_destruction (compiler option). . . . .      | 277 |
| --macro_positions_in_diagnostics (compiler option) . . . . .        | 271 | --no_strtab (ielfdump option) . . . . .                 | 486 |
| --mangled_names_in_messages (linker option) . . . . .               | 306 | --no_system_include (compiler option) . . . . .         | 277 |
| --manual_dynamic_initialization (linker option). . . . .            | 306 | --no_typedefs_in_diagnostics (compiler option). . . . . | 278 |
| --map (linker option). . . . .                                      | 307 | --no_unroll (compiler option) . . . . .                 | 279 |
| --merge_duplicate_sections (linker option). . . . .                 | 307 | --no_utf8_in (ielfdump option) . . . . .                | 487 |
| --mfc (compiler option) . . . . .                                   | 272 | --no_vfe (linker option). . . . .                       | 310 |
| --misrac_verbose (compiler option) . . . . .                        | 253 | --no_warnings (compiler option) . . . . .               | 279 |
| --misrac_verbose (linker option). . . . .                           | 292 | --no_warnings (linker option). . . . .                  | 311 |
| --misrac1998 (compiler option) . . . . .                            | 253 | --no_wrap_diagnostics (compiler option) . . . . .       | 279 |
| --misrac1998 (linker option). . . . .                               | 292 | --no_wrap_diagnostics (linker option) . . . . .         | 311 |
| --misrac2004 (compiler option) . . . . .                            | 253 | --offset (ielftool option) . . . . .                    | 487 |
| --misrac2004 (linker option). . . . .                               | 292 | --only_stdout (compiler option) . . . . .               | 280 |
| --near_const_location (compiler option). . . . .                    | 273 | --only_stdout (linker option). . . . .                  | 311 |
| --nonportable_path_warnings (compiler option). . . . .              | 279 | --option_name (compiler option) . . . . .               | 300 |
| --no_bom (compiler option) . . . . .                                | 273 | --output (compiler option). . . . .                     | 281 |
| --no_bom (ielfdump option) . . . . .                                | 485 | --output (iarchive option) . . . . .                    | 488 |
| --no_bom (iobjmanip option) . . . . .                               | 485 | --output (ielfdump option) . . . . .                    | 488 |
| --no_bom (ismlexport option) . . . . .                              | 485 | --output (linker option) . . . . .                      | 312 |
| --no_call_frame_info (compiler option) . . . . .                    | 274 | --parity (ielftool option) . . . . .                    | 488 |
| --no_clustering (compiler option) . . . . .                         | 274 | --pending_instantiations (compiler option) . . . . .    | 281 |
| --no_code_motion (compiler option) . . . . .                        | 274 | --place_holder (linker option). . . . .                 | 312 |
| --no_cross_call (compiler option). . . . .                          | 275 | --preconfig (linker option) . . . . .                   | 313 |
| --no_cse (compiler option) . . . . .                                | 275 | --predef_macro (compiler option). . . . .               | 281 |
| --no_entry (linker option) . . . . .                                | 308 | --preinclude (compiler option) . . . . .                | 282 |
| --no_fragments (compiler option). . . . .                           | 275 | --preprocess (compiler option) . . . . .                | 282 |
| --no_fragments (linker option) . . . . .                            | 308 | --printf_multibytes (linker option) . . . . .           | 313 |
| --no_free_heap (linker option) . . . . .                            | 309 | --ram_reserve_ranges (ismlexport option) . . . . .      | 489 |
| --no_header (ielfdump option) . . . . .                             | 486 | --range (ielfdump option) . . . . .                     | 490 |
| --no_inline (compiler option) . . . . .                             | 276 | --raw (ielfdump] option). . . . .                       | 490 |

|                                                  |     |                                                             |          |
|--------------------------------------------------|-----|-------------------------------------------------------------|----------|
| --redirect (linker option) . . . . .             | 313 | --text_out (linker option) . . . . .                        | 316      |
| --relaxed_fp (compiler option) . . . . .         | 283 | --threaded_lib (linker option) . . . . .                    | 316      |
| --remarks (compiler option) . . . . .            | 284 | --timezone_lib (linker option) . . . . .                    | 316      |
| --remarks (linker option) . . . . .              | 314 | --tixt (ielftool option) . . . . .                          | 499      |
| --remove_file_path (iobjmanip option) . . . . .  | 491 | --toc (iarchive option) . . . . .                           | 499      |
| --remove_section (iobjmanip option) . . . . .    | 491 | --use_c++_inline (compiler option) . . . . .                | 287      |
| --rename_section (iobjmanip option) . . . . .    | 492 | --use_full_std_template_names (ielfdump option) . . . . .   | 500      |
| --rename_symbol (iobjmanip option) . . . . .     | 492 | --use_full_std_template_names (linker option) . . . . .     | 317      |
| --replace (iarchive option) . . . . .            | 492 | --use_paths_as_written (compiler option) . . . . .          | 287      |
| --require_prototypes (compiler option) . . . . . | 284 | --use_unix_directory_separators (compiler option) . . . . . | 288      |
| --reserve_ranges (ismexport option) . . . . .    | 493 | --verbose (iarchive option) . . . . .                       | 500      |
| --scanf_multibytes (linker option) . . . . .     | 314 | --verbose (ielftool option) . . . . .                       | 500      |
| --search (linker option) . . . . .               | 314 | --version (compiler option) . . . . .                       | 288      |
| --section (ielfdump option) . . . . .            | 494 | --version (linker option) . . . . .                         | 317      |
| --segment (ielfdump option) . . . . .            | 494 | --version (utilities option) . . . . .                      | 501      |
| --self_reloc (ielftool option) . . . . .         | 495 | --vfe (linker option) . . . . .                             | 318      |
| --show_entry_as (ismexport option) . . . . .     | 495 | --vla (compiler option) . . . . .                           | 289      |
| --silent (compiler option) . . . . .             | 284 | --vtoc (iarchive option) . . . . .                          | 501      |
| --silent (iarchive option) . . . . .             | 495 | --warnings_affect_exit_code (compiler option) . . . . .     | 243, 289 |
| --silent (ielftool option) . . . . .             | 495 | --warnings_affect_exit_code (linker option) . . . . .       | 318      |
| --silent (linker option) . . . . .               | 315 | --warnings_are_errors (compiler option) . . . . .           | 289      |
| --simple (ielftool option) . . . . .             | 496 | --warnings_are_errors (linker option) . . . . .             | 318      |
| --simple-ne (ielftool option) . . . . .          | 496 | --warn_about_c_style_casts (compiler option) . . . . .      | 289      |
| --source (ielfdump option) . . . . .             | 496 | --whole_archive (linker option) . . . . .                   | 319      |
| --srec (ielftool option) . . . . .               | 497 | --workseg_area (compiler option) . . . . .                  | 290      |
| --srec-len (ielftool option) . . . . .           | 497 | ? (in reserved identifiers) . . . . .                       | 246      |
| --srec-s3only (ielftool option) . . . . .        | 497 | .bss (ELF section) . . . . .                                | 439      |
| --stack_protection (compiler option) . . . . .   | 285 | .bssf (ELF section) . . . . .                               | 439      |
| --stack_usage_control (linker option) . . . . .  | 315 | .bssf_unit64kp (ELF section) . . . . .                      | 437      |
| --strict (compiler option) . . . . .             | 285 | .bssf.noinit (ELF section) . . . . .                        | 440      |
| --strip (ielftool option) . . . . .              | 498 | .bss.noinit (ELF section) . . . . .                         | 439      |
| --strip (iobjmanip option) . . . . .             | 498 | .callt0 (ELF section) . . . . .                             | 440      |
| --strip (linker option) . . . . .                | 315 | .comment (ELF section) . . . . .                            | 439      |
| --symbols (iarchive option) . . . . .            | 498 | .const (ELF section) . . . . .                              | 440      |
| --system_include_dir (compiler option) . . . . . | 286 | .constf (ELF section) . . . . .                             | 440      |
| --text_out (iarchive option) . . . . .           | 498 | .consth (ELF section) . . . . .                             | 441      |
| --text_out (ielfdump option) . . . . .           | 498 | .data (ELF section) . . . . .                               | 441      |
| --text_out (iobjmanip option) . . . . .          | 498 | .dataf (ELF section) . . . . .                              | 442      |
| --text_out (ismexport option) . . . . .          | 498 | .dataf_init (ELF section) . . . . .                         | 442      |

.data\_init (ELF section) . . . . . 441  
 .data\_unit64kp (ELF section) . . . . . 437  
 .debug (ELF section) . . . . . 438  
 .hbss (ELF section) . . . . . 442  
 .hbss.noinit (ELF section) . . . . . 442  
 .hdata (ELF section) . . . . . 443  
 .hdata\_init (ELF section) . . . . . 443  
 .iar.debug (ELF section) . . . . . 439  
 .iar.dynexit (ELF section) . . . . . 443  
 .init\_array (section) . . . . . 444  
 .intvec (section) . . . . . 444  
 .option\_byte (section) . . . . . 444  
 .preinit\_array (section) . . . . . 445  
 .rel (ELF section) . . . . . 439  
 .rela (ELF section) . . . . . 439  
 .sbss (ELF section) . . . . . 445  
 .sbss.noinit (ELF section) . . . . . 445  
 .sdata (ELF section) . . . . . 445  
 .sdata\_init (ELF section) . . . . . 446  
 .security\_id (section) . . . . . 446  
 .shstrtab (ELF section) . . . . . 439  
 .strtab (ELF section) . . . . . 439  
 .switch (section) . . . . . 446  
 .switchf (section) . . . . . 446  
 .symtab (ELF section) . . . . . 439  
 .text (ELF section) . . . . . 446  
 .textf (ELF section) . . . . . 447  
 .text\_unit64kp (ELF section) . . . . . 438  
 .vector (ELF section) . . . . . 438  
 .wrkseg (ELF section) . . . . . 447  
 @ (operator)  
   placing at absolute address . . . . . 223  
   placing in sections . . . . . 224  
 #include directive,  
 implementation-defined behavior for C++ . . . . . 512  
 #include files, specifying . . . . . 241, 270  
 #include\_next . . . . . 190  
 #pragma directive . . . . . 353  
   implementation-defined behavior for C++ . . . . . 512  
 #warning . . . . . 190

#warning message (preprocessor extension) . . . . . 392  
 %Z replacement string,  
 implementation-defined behavior in C . . . . . 538  
 \$Sub\$\$ pattern . . . . . 218  
 \$Super\$\$ pattern . . . . . 218  
 \$\$ (in reserved identifiers) . . . . . 246

## Numerics

16-bit pointers, accessing memory . . . . . 72  
 24-bit pointers, accessing memory . . . . . 73  
 32-bits (floating-point format) . . . . . 327  
 64-bit data types, avoiding . . . . . 219  
 64-bits (floating-point format) . . . . . 327