

# IAR Embedded Workbench<sup>®</sup>

## IAR C/C++ Compiler User Guide

for Atmel<sup>®</sup> Corporation's  
**AVR32 Microprocessor Family**



## **COPYRIGHT NOTICE**

© 2002–2015 IAR Systems AB.

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

## **DISCLAIMER**

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

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

## **TRADEMARKS**

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Atmel is a registered trademark of Atmel® Corporation. AVR is a registered trademark and AVR32 is a trademark, both of Atmel® 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**

Third edition: April 2015

Part number: CAVR32-3

This guide applies to version 4.x of IAR Embedded Workbench® for Atmel® Corporation's AVR32 microprocessor family.

Internal reference: M18, csrct2010.1, V\_110411, IMAE.

# Brief contents

Tables .....	27
Preface .....	29
<b>Part 1. Using the compiler</b> .....	<b>37</b>
Introduction to the IAR build tools .....	39
Developing embedded applications .....	45
Data storage .....	59
Functions .....	71
Linking overview .....	89
Linking your application .....	105
The DLIB runtime environment .....	119
Assembler language interface .....	159
Using C .....	187
Using C++ .....	195
Application-related considerations .....	205
Efficient coding for embedded applications .....	211
<b>Part 2. Reference information</b> .....	<b>229</b>
External interface details .....	231
Compiler options .....	237
Data representation .....	275
Extended keywords .....	289
Pragma directives .....	305

Intrinsic functions .....	329
The preprocessor .....	347
Library functions .....	355
Segment reference .....	365
The stack usage control file .....	383
Implementation-defined behavior for Standard C .....	391
Implementation-defined behavior for C89 .....	407
Index .....	419

# Contents

Tables .....	27
Preface .....	29
<b>Who should read this guide</b> .....	29
Required knowledge .....	29
<b>How to use this guide</b> .....	29
<b>What this guide contains</b> .....	30
Part 1. Using the compiler .....	30
Part 2. Reference information .....	30
<b>Other documentation</b> .....	31
User and reference guides .....	31
The online help system .....	32
Further reading .....	33
Web sites .....	33
<b>Document conventions</b> .....	34
Typographic conventions .....	34
Naming conventions .....	35
<b>Part I. Using the compiler</b> .....	37
Introduction to the IAR build tools .....	39
<b>The IAR build tools—an overview</b> .....	39
IAR C/C++ Compiler .....	39
IAR Assembler .....	40
The IAR XLINK Linker .....	40
External tools .....	40
<b>IAR language overview</b> .....	40
<b>Device support</b> .....	41
Supported AVR32 devices .....	41
Preconfigured support files .....	41
Examples for getting started .....	42

<b>Special support for embedded systems</b> .....	42
Extended keywords .....	42
Pragma directives .....	42
Predefined symbols .....	43
Accessing low-level features .....	43
<b>Developing embedded applications</b> .....	45
<b>Developing embedded software using IAR build tools</b> .....	45
CPU features and constraints .....	45
Mapping of memory .....	45
Communication with peripheral units .....	46
Event handling .....	46
System startup .....	46
Real-time operating systems .....	47
<b>The build process—an overview</b> .....	47
The translation process .....	47
The linking process .....	48
After linking .....	49
<b>Application execution—an overview</b> .....	50
The initialization phase .....	50
The execution phase .....	53
The termination phase .....	53
<b>Building applications—an overview</b> .....	54
<b>Basic project configuration</b> .....	54
Processor configuration .....	55
Data model .....	56
Code model .....	56
Optimization for speed and size .....	57
Runtime environment .....	57
<b>Data storage</b> .....	59
<b>Introduction</b> .....	59
Different ways to store data .....	59
<b>Memory types</b> .....	60
Introduction to memory types .....	60

Using data memory attributes .....	63
Structures and memory types .....	65
More examples .....	65
C++ and memory types .....	66
<b>Data models .....</b>	<b>66</b>
Specifying a data model .....	67
<b>Storage of auto variables and parameters .....</b>	<b>68</b>
The stack .....	68
<b>Dynamic memory on the heap .....</b>	<b>69</b>
Potential problems .....	69
<b>Functions .....</b>	<b>71</b>
<b>Function-related extensions .....</b>	<b>71</b>
<b>Code models and memory attributes for function storage .....</b>	<b>71</b>
Using function memory attributes .....	72
<b>Primitives for interrupts, concurrency, and OS-related programming .....</b>	<b>73</b>
Interrupt functions .....	73
Exception handlers .....	75
ACALL functions .....	77
SCALL functions .....	78
Monitor functions .....	79
<b>Execution in RAM .....</b>	<b>83</b>
<b>Implementing middleware using FlashVault™ .....</b>	<b>84</b>
Implementing a single entry point API .....	85
Implementing a multiple entry point API .....	85
Locking down the firmware at download .....	86
<b>Inlining functions .....</b>	<b>87</b>
C versus C++ semantics .....	87
Features controlling function inlining .....	88
<b>Linking overview .....</b>	<b>89</b>
<b>Linking—an overview .....</b>	<b>89</b>
<b>Segments and memory .....</b>	<b>90</b>
What is a segment? .....	90

<b>The linking process in detail</b> .....	91
<b>Placing code and data—the linker configuration file</b> .....	92
The contents of the linker configuration file .....	93
<b>Initialization at system startup</b> .....	93
Static data memory segments .....	94
The initialization process .....	95
<b>Stack usage analysis</b> .....	95
Introduction to stack usage analysis .....	95
Performing a stack usage analysis .....	96
Result of an analysis—the map file contents .....	97
Specifying additional stack usage information .....	99
Limitations .....	100
Situations where warnings are issued .....	101
Call graph log .....	101
<b>Linking your application</b> .....	105
<b>Linking considerations</b> .....	105
Placing segments .....	106
Placing data .....	108
Setting up stack memory .....	109
Setting up heap memory .....	109
Placing code .....	110
Keeping modules .....	113
Keeping symbols and segments .....	113
Application startup .....	113
Interaction between XLINK and your application .....	113
Producing other output formats than UBROF .....	114
<b>Linking for segment-translated systems</b> .....	114
Segment-translated mode .....	115
Implications for the linker configuration file .....	115
Mapped memories .....	116
<b>Verifying the linked result of code and data placement</b> .....	117
Segment too long errors and range errors .....	117
Linker map file .....	118



The DLIB runtime environment .....	119
<b>Introduction to the runtime environment</b> .....	119
Runtime environment functionality .....	119
Setting up the runtime environment .....	120
<b>Using prebuilt libraries</b> .....	121
Choosing a library .....	122
Library filename syntax .....	122
Customizing a prebuilt library without rebuilding .....	122
<b>Choosing formatters for printf and scanf</b> .....	123
Choosing a printf formatter .....	123
Choosing a scanf formatter .....	124
<b>Application debug support</b> .....	125
Including C-SPY debugging support .....	126
The debug library functionality .....	126
The C-SPY Terminal I/O window .....	127
Low-level functions in the debug library .....	128
<b>Adapting the library for target hardware</b> .....	128
Library low-level interface .....	129
<b>Overriding library modules</b> .....	129
<b>Building and using a customized library</b> .....	130
Setting up a library project .....	130
Modifying the library functionality .....	131
Using a customized library .....	131
<b>System startup and termination</b> .....	131
System startup .....	132
System termination .....	134
<b>Customizing system initialization</b> .....	135
__low_level_init .....	135
Modifying the file cstartup.s82 .....	136
<b>Library configurations</b> .....	136
Choosing a runtime configuration .....	136
<b>Standard streams for input and output</b> .....	137
Implementing low-level character input and output .....	137

<b>Configuration symbols for printf and scanf</b> .....	139
Customizing formatting capabilities .....	140
<b>File input and output</b> .....	141
<b>Locale</b> .....	141
Locale support in prebuilt libraries .....	142
Customizing the locale support .....	142
Changing locales at runtime .....	143
<b>Environment interaction</b> .....	144
The getenv function .....	144
The system function .....	144
<b>Signal and raise</b> .....	145
<b>Time</b> .....	145
<b>Strtod</b> .....	146
<b>Math functions</b> .....	146
Smaller versions .....	146
More accurate versions .....	147
<b>Assert</b> .....	148
<b>Managing a multithreaded environment</b> .....	149
Multithread support in the DLIB library .....	149
Enabling multithread support .....	150
TLS in the linker configuration file .....	154
<b>Checking module consistency</b> .....	154
Runtime model attributes .....	154
Using runtime model attributes .....	155
Predefined runtime attributes .....	156
<b>Assembler language interface</b> .....	159
<b>Mixing C and assembler</b> .....	159
Intrinsic functions .....	159
Mixing C and assembler modules .....	160
Inline assembler .....	160
Reference information for inline assembler .....	162
An example of how to use clobbered memory .....	167

<b>Calling assembler routines from C</b> .....	167
Creating skeleton code .....	167
Compiling the skeleton code .....	168
<b>Calling assembler routines from C++</b> .....	169
<b>Calling convention</b> .....	170
Function declarations .....	171
Using C linkage in C++ source code .....	171
Preserved versus scratch registers .....	172
Function entrance .....	173
Function exit .....	175
Calls in supervisor mode .....	176
Alternative calling convention for FlashVault implementation functions .....	176
Examples .....	177
Function directives .....	178
<b>Assembler instructions used for calling functions</b> .....	178
Calling functions in the Small and medium code models .....	179
Calling functions in the Large code model .....	179
<b>Memory access methods</b> .....	180
The main memory access method (data21, data32) .....	181
Read-modify-write access method (data17) .....	181
The system and debug register access method .....	181
<b>Call frame information</b> .....	181
CFI directives .....	182
Creating assembler source with CFI support .....	183
<b>Using C</b> .....	187
<b>C language overview</b> .....	187
<b>Extensions overview</b> .....	188
Enabling language extensions .....	189
<b>IAR C language extensions</b> .....	189
Extensions for embedded systems programming .....	190
Relaxations to Standard C .....	192

Using C++ .....	195
<b>Overview—EC++ and EEC++</b> .....	195
Embedded C++ .....	195
Extended Embedded C++ .....	196
<b>Enabling support for C++</b> .....	197
<b>EC++ feature descriptions</b> .....	197
Using IAR attributes with Classes .....	197
Function types .....	198
Using static class objects in interrupts .....	199
Using New handlers .....	199
Templates .....	199
Debug support in C-SPY .....	199
<b>EEC++ feature description</b> .....	200
Templates .....	200
Variants of cast operators .....	200
Mutable .....	200
Namespace .....	200
The STD namespace .....	200
<b>C++ language extensions</b> .....	201
Application-related considerations .....	205
<b>Stack considerations</b> .....	205
Stack size considerations .....	205
<b>Heap considerations</b> .....	205
Heap segments in DLIB .....	206
Heap size and standard I/O .....	206
<b>Interaction between the tools and your application</b> .....	206
<b>Checksum calculation</b> .....	207
Calculating a checksum .....	208
Adding a checksum function to your source code .....	208
Things to remember .....	209

Efficient coding for embedded applications .....	211
<b>Selecting data types</b> .....	211
Using efficient data types .....	211
Floating-point types .....	211
Alignment of elements in a structure .....	212
Anonymous structs and unions .....	213
<b>Controlling data and function placement in memory</b> .....	214
Data placement at an absolute location .....	215
Data and function placement in segments .....	217
<b>Controlling compiler optimizations</b> .....	218
Scope for performed optimizations .....	218
Multi-file compilation units .....	219
Optimization levels .....	220
Speed versus size .....	221
Fine-tuning enabled transformations .....	221
<b>Facilitating good code generation</b> .....	223
Writing optimization-friendly source code .....	224
Saving stack space and RAM memory .....	224
Function prototypes .....	224
Integer types and bit negation .....	225
Protecting simultaneously accessed variables .....	226
Accessing special function registers .....	227
Passing values between C and assembler objects .....	228
Non-initialized variables .....	228
<b>Part 2. Reference information</b> .....	229
External interface details .....	231
<b>Invocation syntax</b> .....	231
Compiler invocation syntax .....	231
Passing options .....	232
Environment variables .....	232
<b>Include file search procedure</b> .....	232

<b>Compiler output</b> .....	233
Error return codes .....	234
<b>Diagnostics</b> .....	235
Message format .....	235
Severity levels .....	235
Setting the severity level .....	236
Internal error .....	236
<b>Compiler options</b> .....	237
<b>Options syntax</b> .....	237
Types of options .....	237
Rules for specifying parameters .....	237
<b>Summary of compiler options</b> .....	239
<b>Descriptions of compiler options</b> .....	243
--avr32_dsp_instructions .....	243
--avr32_flashvault .....	243
--avr32_fpu_instructions .....	244
--avr32_rmw_instructions .....	244
--avr32_simd_instructions .....	245
--c89 .....	245
--char_is_signed .....	246
--char_is_unsigned .....	246
--code_model .....	246
--core .....	247
--core_revision .....	247
--cpu .....	247
--cpu_info .....	248
-D .....	248
--data_model .....	249
--debug, -r .....	249
--dependencies .....	250
--diag_error .....	251
--diag_remark .....	251
--diag_suppress .....	251

--diag_warning .....	252
--diagnostics_tables .....	252
--disable_inline_asm_label_replacement .....	253
--discard_unused_publics .....	253
--dlib_config .....	253
-e .....	254
--ec++ .....	255
--eec++ .....	255
--enable_multibytes .....	255
--enable_restrict .....	256
--error_limit .....	256
-f .....	256
--fp_implementation .....	257
--guard_calls .....	257
--header_context .....	257
-I .....	258
-l .....	258
--library_module .....	259
--macro_positions_in_diagnostics .....	259
--mfc .....	260
--minimize_constant_tables .....	260
--module_name .....	260
--no_clustering .....	261
--no_code_motion .....	261
--no_cse .....	261
--no_inline .....	262
--no_path_in_file_macros .....	262
--no_scheduling .....	262
--no_size_constraints .....	263
--no_static_destruction .....	263
--no_system_include .....	263
--no_tbaa .....	264
--no_typedefs_in_diagnostics .....	264
--no_unroll .....	264

--no_warnings	265
--no_wrap_diagnostics	265
-O	265
--omit_types	266
--only_stdout	266
--output, -o	266
--pending_instantiations	267
--predef_macros	267
--preinclude	268
--preprocess	268
--public_equ	268
--relaxed_fp	269
--remarks	269
--require_prototypes	270
--silent	270
--strict	270
--system_include_dir	271
--unaligned_word_access	271
--use_cplusplus_inline	272
--variable_enum_size	272
--vla	272
--warn_about_c_style_casts	273
--warnings_affect_exit_code	273
--warnings_are_errors	273

Data representation ..... 275

**Alignment** ..... 275

**Basic data types—integer types** ..... 276

Integer types—an overview	276
Bool	276
The enum type	276
The char type	277
The wchar_t type	277
Bitfields	277



<b>Basic data types—floating-point types</b> .....	279
Floating-point environment .....	280
32-bit floating-point format .....	280
64-bit floating-point format .....	280
Representation of special floating-point numbers .....	280
<b>Pointer types</b> .....	281
Function pointers .....	281
Data pointers .....	281
Casting .....	282
<b>Structure types</b> .....	282
Alignment of structure types .....	283
General layout .....	283
Packed structure types .....	283
<b>Type qualifiers</b> .....	284
Declaring objects volatile .....	284
Declaring objects volatile and const .....	286
Declaring objects const .....	286
<b>Data types in C++</b> .....	287
<b>Extended keywords</b> .....	289
<b>General syntax rules for extended keywords</b> .....	289
Type attributes .....	289
Object attributes .....	292
<b>Summary of extended keywords</b> .....	292
<b>Descriptions of extended keywords</b> .....	293
__acall .....	293
__code21 .....	294
__code32 .....	294
__data17 .....	295
__data21 .....	295
__data32 .....	296
__dbgreg .....	296
__exception .....	296
__flashvault .....	297

__flashvault_impl .....	297
__imported .....	298
__interrupt .....	298
__intrinsic .....	298
__monitor .....	298
__nested .....	299
__no_alloc, __no_alloc16 .....	299
__no_alloc_str, __no_alloc_str16 .....	300
__no_init .....	300
__noreturn .....	301
__packed .....	301
__ramfunc .....	302
__root .....	303
__scall .....	303
__sysreg .....	303
<b>Pragma directives .....</b>	<b>305</b>
<b>Summary of pragma directives .....</b>	<b>305</b>
<b>Descriptions of pragma directives .....</b>	<b>307</b>
bitfields .....	307
calls .....	308
call_graph_root .....	308
constseg .....	309
data_alignment .....	309
dataseg .....	310
default_function_attributes .....	310
default_variable_attributes .....	311
diag_default .....	312
diag_error .....	313
diag_remark .....	313
diag_suppress .....	313
diag_warning .....	314
error .....	314
exception .....	314

flashvault_vector .....	315
handler .....	315
include_alias .....	316
inline .....	317
language .....	317
location .....	318
message .....	319
object_attribute .....	319
optimize .....	320
pack .....	321
__printf_args .....	322
public_equ .....	322
required .....	322
rtmodel .....	323
__scanf_args .....	324
segment .....	324
shadow_registers .....	325
STDC CX_LIMITED_RANGE .....	326
STDC FENV_ACCESS .....	326
STDC FP_CONTRACT .....	327
type_attribute .....	327
vector .....	328
<b>Intrinsic functions</b> .....	<b>329</b>
<b>Summary of intrinsic functions</b> .....	<b>329</b>
Intrinsic inline functions .....	329
Summary and description of ETSI functions .....	331
<b>Descriptions of intrinsic functions</b> .....	<b>333</b>
__bit_reverse .....	333
__BREAKPOINT .....	333
__cache_control .....	334
__clear_status_flag .....	334
__COP .....	334
__COP_get_register32 .....	335

__COP_get_register64 .....	335
__COP_get_registers .....	335
__COP_set_registers .....	336
__COP_set_register32 .....	336
__COP_set_register64 .....	336
__count_leading_zeros .....	336
__count_trailing_zeros .....	337
__disable_interrupt .....	337
__enable_interrupt .....	337
__exchange_memory .....	337
__get_debug_register .....	338
__get_interrupt_state .....	338
__get_system_register .....	339
__get_user_context .....	339
__max .....	339
__min .....	339
__no_operation .....	340
__prefetch_cache .....	340
__read_TLB_entry .....	340
__search_TLB_entry .....	340
__set_debug_register .....	341
__set_interrupt_state .....	341
__set_status_flag .....	341
__set_system_register .....	342
__set_user_context .....	342
__signed_saturate .....	343
__sleep .....	343
__store_conditional .....	343
__swap_bytes .....	344
__swap_bytes_in_halfwords .....	344
__swap_halfwords .....	344
__synchronize_write_buffer .....	345
__test_status_flag .....	345
__unsigned_saturate .....	345

__write_TLB_entry .....	346
The preprocessor .....	347
<b>Overview of the preprocessor</b> .....	347
<b>Description of predefined preprocessor symbols</b> .....	348
__BASE_FILE__ .....	348
__BUILD_NUMBER__ .....	348
__CODE_MODEL__ .....	348
__CORE__ .....	348
__CORE_REVISION__ .....	348
__COUNTER__ .....	348
__cplusplus .....	349
__DATA_MODEL__ .....	349
__DATE__ .....	349
__DEFAULT_CODE_SEGMENT__ .....	349
__DEFAULT_CONST_SEGMENT__ .....	349
__DEFAULT_DATA_SEGMENT__ .....	349
__embedded_cplusplus .....	349
__FILE__ .....	350
__func__ .....	350
__FUNCTION__ .....	350
__HAS_DSP_INSTRUCTIONS__ .....	350
__HAS_FPU_INSTRUCTIONS__ .....	350
__HAS_RMW_INSTRUCTIONS__ .....	351
__HAS_SIMD_INSTRUCTIONS__ .....	351
__IAR_SYSTEMS_ICC__ .....	351
__ICCavr32__ .....	351
__LINE__ .....	351
__PART__ .....	351
__PRETTY_FUNCTION__ .....	352
__STDC__ .....	352
__STDC_VERSION__ .....	352
__SUBVERSION__ .....	352
__TIME__ .....	352

__TIMESTAMP__ .....	352
__VER__ .....	353
<b>Descriptions of miscellaneous preprocessor extensions</b> .....	353
NDEBUG .....	353
#warning message .....	353
<b>Library functions</b> .....	355
<b>Library overview</b> .....	355
Header files .....	355
Library object files .....	355
Alternative more accurate library functions .....	356
Reentrancy .....	356
The longjmp function .....	357
<b>IAR DLIB Library</b> .....	357
C header files .....	357
C++ header files .....	358
Library functions as intrinsic functions .....	361
Added C functionality .....	361
Symbols used internally by the library .....	362
<b>Segment reference</b> .....	365
<b>Summary of segments</b> .....	365
<b>Descriptions of segments</b> .....	367
ACTAB .....	367
CHECKSUM .....	368
CODE21 .....	368
CODE32 .....	368
CSTACK .....	369
DATA17_AC .....	369
DATA17_AN .....	369
DATA17_C .....	369
DATA17_I .....	370
DATA17_ID .....	370
DATA17_N .....	370
DATA17_Z .....	371

DATA21_AC .....	371
DATA21_AN .....	371
DATA21_C .....	371
DATA21_I .....	372
DATA21_ID .....	372
DATA21_N .....	372
DATA21_Z .....	373
DATA32_AC .....	373
DATA32_AN .....	373
DATA32_C .....	373
DATA32_I .....	374
DATA32_ID .....	374
DATA32_N .....	374
DATA32_Z .....	375
DBGREG_AC .....	375
DBGREG_AN .....	375
DIFUNCT .....	375
EVBYTES1 .....	376
EVBYTES2 .....	376
EVBYTES3 .....	376
EVSEG .....	377
EVTAB .....	377
EV100 .....	377
FVVEC .....	378
HEAP .....	378
HTAB .....	378
INITTAB .....	379
RAMCODE21 .....	379
RAMCODE21_ID .....	380
RAMCODE32 .....	380
RAMCODE32_ID .....	380
RESET .....	381
RESETCODE .....	381
SSTACK .....	381

SWITCH .....	382
SYSREG_AC .....	382
SYSREG_AN .....	382
TRACEBUFFER .....	382
The stack usage control file .....	383
<b>Overview</b> .....	383
C++ names .....	383
<b>Stack usage control directives</b> .....	383
call graph root directive .....	383
check that directive .....	384
exclude directive .....	384
function directive .....	385
max recursion depth directive .....	385
no calls from directive .....	386
possible calls directive .....	386
<b>Syntactic components</b> .....	387
<i>category</i> .....	387
<i>func-spec</i> .....	387
<i>module-spec</i> .....	387
<i>name</i> .....	388
<i>call-info</i> .....	388
<i>stack-size</i> .....	389
<i>size</i> .....	389
Implementation-defined behavior for Standard C .....	391
<b>Descriptions of implementation-defined behavior</b> .....	391
J.3.1 Translation .....	391
J.3.2 Environment .....	392
J.3.3 Identifiers .....	393
J.3.4 Characters .....	393
J.3.5 Integers .....	394
J.3.6 Floating point .....	395
J.3.7 Arrays and pointers .....	396
J.3.8 Hints .....	396



J.3.9 Structures, unions, enumerations, and bitfields .....	397
J.3.10 Qualifiers .....	397
J.3.11 Preprocessing directives .....	397
J.3.12 Library functions .....	399
J.3.13 Architecture .....	404
J.4 Locale .....	404
<b>Implementation-defined behavior for C89 .....</b>	<b>407</b>
<b>Descriptions of implementation-defined behavior .....</b>	<b>407</b>
Translation .....	407
Environment .....	407
Identifiers .....	408
Characters .....	408
Integers .....	409
Floating point .....	410
Arrays and pointers .....	410
Registers .....	411
Structures, unions, enumerations, and bitfields .....	411
Qualifiers .....	411
Declarators .....	412
Statements .....	412
Preprocessing directives .....	412
IAR DLIB Library functions .....	414
<b>Index .....</b>	<b>419</b>



# Tables

1: Typographic conventions used in this guide .....	34
2: Naming conventions used in this guide .....	35
3: Memory types and their corresponding memory attributes .....	63
4: Data model characteristics .....	67
5: Function memory attributes .....	72
6: segments holding initialized data .....	94
7: Mapped memory, example of .....	117
8: Customizable items .....	123
9: Formatters for printf .....	124
10: Formatters for scanf .....	125
11: Levels of debugging support in runtime libraries .....	126
12: Functions with special meanings when linked with debug library .....	128
13: Library configurations .....	136
14: Descriptions of printf configuration symbols .....	140
15: Descriptions of scanf configuration symbols .....	140
16: Low-level I/O files .....	141
17: Library objects using TLS .....	149
18: Macros for implementing TLS allocation .....	152
19: Example of runtime model attributes .....	154
20: Predefined runtime model attributes .....	156
21: Inline assembler operand constraints .....	163
22: Supported constraint modifiers .....	164
23: Operand modifiers and transformations .....	165
24: List of valid clobbers .....	166
25: Registers used for passing parameters .....	174
26: Registers used for returning values .....	175
27: Call frame information resources defined in a names block .....	182
28: Language extensions .....	189
29: Compiler optimization levels .....	220
30: Compiler environment variables .....	232
31: Error return codes .....	234

32: Compiler options summary .....	239
33: Integer types .....	276
34: Floating-point types .....	279
35: Function pointers .....	281
36: Data pointers .....	281
37: Volatile accesses .....	286
38: Extended keywords summary .....	292
39: Pragma directives summary .....	305
40: Intrinsic functions summary .....	329
41: ETSI functions summary .....	331
42: Traditional Standard C header files—DLIB .....	357
43: C++ header files .....	359
44: Standard template library header files .....	359
45: New Standard C header files—DLIB .....	360
46: Segment summary .....	365
47: Message returned by <code>strerror()</code> —IAR DLIB library .....	406
48: Message returned by <code>strerror()</code> —IAR DLIB library .....	417

# Preface

Welcome to the *IAR C/C++ Compiler User Guide for AVR32*. The purpose of this guide is to provide you with detailed reference information that can help you to use the compiler to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

---

## Who should read this guide

Read this guide if you plan to develop an application using the C or C++ language for the AVR32 microprocessor and need detailed reference information on how to use the compiler. You should have working knowledge of:

### REQUIRED KNOWLEDGE

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

- The architecture and instruction set of the AVR32 microprocessor (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

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

---

## How to use this guide

When you start using the IAR C/C++ Compiler for AVR32, you should read *Part 1. Using the compiler* in this guide.

When you are familiar with the compiler and have already configured your project, you can focus more on *Part 2. Reference information*.

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

---

## What this guide contains

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

### PART I. USING THE COMPILER

- *Introduction to the IAR build tools* gives an introduction to the IAR build tools, which includes an overview of the tools, the programming languages, the available device support, and extensions provided for supporting specific features of the AVR32 microprocessor.
- *Developing embedded applications* gives the information you need to get started developing your embedded software using the IAR build tools.
- *Data storage* describes how to store data in memory.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Linking overview* describes the linking process using the IAR XLINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using XLINK options and tailoring the linker configuration file.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Application-related considerations* discusses a selected range of application issues related to using the compiler and linker.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

### PART 2. REFERENCE INFORMATION

- *External interface details* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the

different types of compiler output. The chapter also describes how the compiler's diagnostic system works.

- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Extended keywords* gives reference information about each of the AVR32-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 AVR32-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *The stack usage control file* describes the syntax and semantics of stack usage control files.
- *Implementation-defined behavior for Standard C* describes how the compiler handles the implementation-defined areas of Standard C.
- *Implementation-defined behavior for C89* describes how the compiler handles the implementation-defined areas of the C language standard C89.

---

## Other documentation

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

### USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.

- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for AVR32*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for AVR32*.
- Programming for the IAR C/C++ Compiler for AVR32, is available in the *IAR C/C++ Compiler User Guide for AVR32*.
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, is available in the IAR Linker and Library Tools Reference Guide.
- Programming for the IAR Assembler for AVR32, is available in the *IAR Assembler User Guide for AVR32*.
- Using the IAR DLIB Library, is available in the *DLIB Library Reference information*, available in the online help system.
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for AVR32, is available in the *IAR Embedded Workbench® Migration Guide*.

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

## THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about project management, editing, and building in the IDE
- Information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.



## FURTHER READING

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

- *Atmel® AVR32 Architecture document* provided by Atmel® Corporation.
- *Atmel® AVR32UC Technical Reference Manual* provided by Atmel® Corporation.
- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.
- Meyers, Scott. *Effective STL*. Addison-Wesley.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++*. Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.

## WEB SITES

Recommended web sites:

- The Atmel® Corporation web site, [www.atmel.com](http://www.atmel.com), that contains information and news about the Atmel AVR32 microprocessors.
- The IAR Systems web site, [www.iar.com](http://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](http://www.open-std.org/jtc1/sc22/wg14).
- The web site of the C++ Standards Committee, [www.open-std.org/jtc1/sc22/wg21](http://www.open-std.org/jtc1/sc22/wg21).

- Finally, the Embedded C++ Technical Committee web site, [www.caravan.net/ec2plus](http://www.caravan.net/ec2plus), that contains information about the Embedded C++ standard.

---

## Document conventions

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

When referring to a directory in your product installation, for example `avr32\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 7.n\avr32\doc`.

### TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:



Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> <li>• Source code examples and file paths.</li> <li>• Text on the command line.</li> <li>• Binary, hexadecimal, and octal numbers.</li> </ul>
<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 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 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.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> <li>• A cross-reference within this guide or to another guide.</li> <li>• Emphasis.</li> </ul>
...	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.

Table 1: Typographic conventions used in this guide



Style	Used for
	Identifies helpful tips and programming hints.
	Identifies warnings.

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

## NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for AVR32	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for AVR32	the IDE
IAR C-SPY® Debugger for AVR32	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for AVR32	the compiler
IAR Assembler™ for AVR32	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library

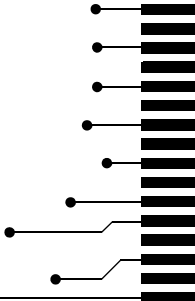
Table 2: *Naming conventions used in this guide*



# Part I. Using the compiler

This part of the *IAR C/C++ Compiler User Guide for AVR32* includes these chapters:

- Introduction to the IAR build tools
- Developing embedded applications
- Data storage
- Functions
- Linking overview
- 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 AVR32-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.



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

For information about the IDE, see the *IDE Project Management and Building Guide for AVR32*.



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

### IAR C/C++ COMPILER

The IAR C/C++ Compiler for AVR32 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 AVR32-specific facilities.

## IAR ASSEMBLER

The IAR Assembler for AVR32 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 AVR32 uses the same mnemonics and operand syntax as the Atmel® Corporation AVR32 Assembler, which simplifies the migration of existing code. For more information, see the *IAR Assembler User Guide for AVR32*.

## THE IAR XLINK LINKER

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

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

## EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IDE Project Management and Building Guide for AVR32*.

---

# IAR language overview

The IAR C/C++ Compiler for AVR32 supports:

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



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

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

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

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *IAR Assembler User Guide for AVR32*.

---

## 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 AVR32 DEVICES

The IAR C/C++ Compiler for AVR32 supports all devices based on the standard Atmel® Corporation AVR32 cores, such as the UC3 and AP7 families

### PRECONFIGURED SUPPORT FILES

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

### Header files for I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension `.h`. The product package supplies I/O files for all publicly available devices at the time of the product release.

### Linker configuration files

The `avr32\config` directory contains ready-made linker configuration files for all supported devices. The files have the filename extension `.xcl` and contain the information required by the linker. For more information about the linker configuration file, see *Placing code and data—the linker configuration file*, page 92 as well as the *IAR Linker and Library Tools Reference Guide*.

### Device description files

The debugger handles several of the device-specific requirements, such as definitions of available memory areas, peripheral registers and groups of these, by using device description files. These files are located in the `avr32\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 AVR32*.

### EXAMPLES FOR GETTING STARTED

The `avr32\examples` directory contains several hundreds of examples of working applications to give you a smooth start with your development. The complexity of the examples ranges from simple LED blink to USB mass storage controllers. Examples are provided for most of the supported devices.

---

## Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the AVR32 microprocessor.

### EXTENDED KEYWORDS

The compiler provides a set of keywords that can be used for configuring how the code is generated. For example, there are keywords for controlling the memory type for individual variables as well as for declaring special function types.

By default, language extensions are enabled in the IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, `-e`, page 254 for additional information.

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

### PRAGMA DIRECTIVES

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

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

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

## PREDEFINED SYMBOLS

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

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

## ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 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 your help, you have compiler options, extended keywords, pragma directives, etc.

### CPU FEATURES AND CONSTRAINTS

Some of the basic features of the AVR32 microprocessor are:

- DSP instruction set extension
- FPU instruction set extension
- RMW instruction set extension
- SIMD instruction set extension
- FlashVault/Secure State extension

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

### MAPPING OF MEMORY

Embedded systems typically contain various types of memory, such as 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 accessed seldom but must maintain their value after power off, so they should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. For more information, see *Controlling data and function placement in memory*, page 214. 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 92.

## COMMUNICATION WITH PERIPHERAL UNITS

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

Standard peripheral units are defined in device-specific I/O header files with the filename extension `.h`. See *Device support*, page 41. For an example, see *Accessing special function registers*, page 227.

## 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 microprocessor immediately stops executing the code it runs, and starts executing an interrupt routine instead.

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

## SYSTEM STARTUP

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

As an embedded software developer, you must ensure that the startup code is located at the dedicated memory addresses, or can be accessed using a pointer from the vector

table. This means that startup code and the initial vector table must be placed in non-volatile memory, such as ROM, EPROM, or flash.

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

## REAL-TIME OPERATING SYSTEMS

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

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

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

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

---

## The build process—an overview

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

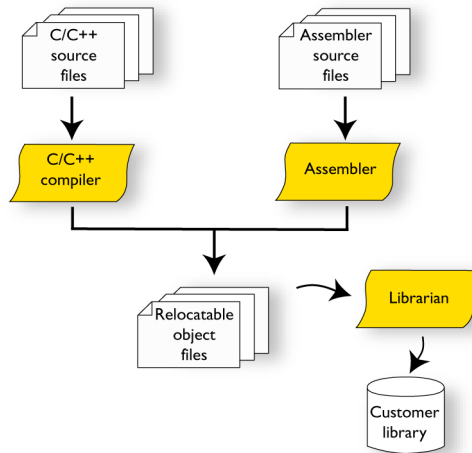
To get familiar with the process in practice, you should run one or more of the tutorials available from the IAR Information Center.

## THE TRANSLATION PROCESS

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

**Note:** The compiler can also be used for translating C/C++ source code into assembler source code. If required, you can modify the assembler source code which then can be assembled into object code. For more information about the IAR Assembler, see the *IAR Assembler User Guide for AVR32*.

This illustration shows the translation process:



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

## THE LINKING PROCESS

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

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

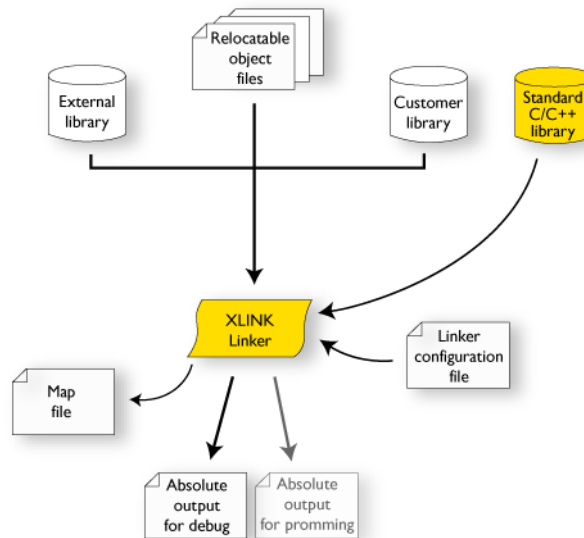
- Several object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label (set by default)
- The linker configuration file that describes placement of code and data in the memory of the target system
- Information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

The IAR XLINK Linker produces output according to your specifications. Choose the output format that suits your purpose. You might want to load the output to a



debugger—which means that you need output with debug information. Alternatively, you might want to load output to a flash loader or a PROM programmer—in which case you need output without debug information, such as Intel hex or Motorola S-records. The option `-F` can be used for specifying the output format.

This illustration shows the linking process:



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

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

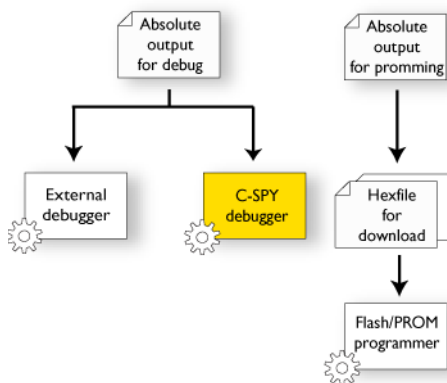
For more information about the procedure performed by the linker, see the *IAR Linker and Library Tools Reference Guide*.

## AFTER LINKING

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

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

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



## Application execution—an overview

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

- Initialization phase
- Execution phase
- Termination phase.

### THE INITIALIZATION PHASE

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

- Hardware initialization, which generally at least initializes the stack pointer.  
The hardware initialization is typically performed in the system startup code `cstartup.s82` and if required, by an extra low-level routine that you provide. It might include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.
- Software C/C++ system initialization  
Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.
- Application initialization

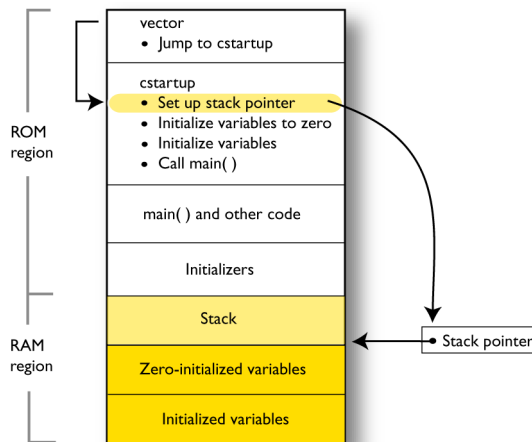
This depends entirely on your application. It can include setting up an RTOS kernel and starting initial tasks for an RTOS-driven application. For a bare-bone application,

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

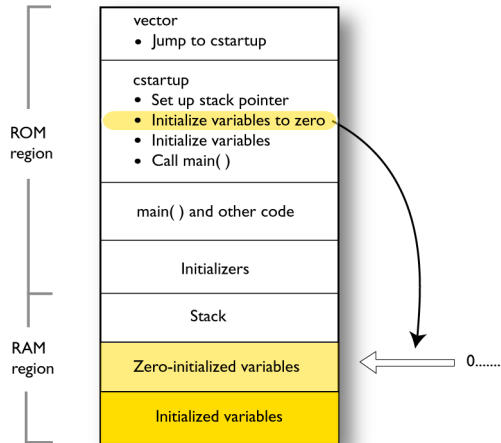
For a ROM/flash-based system, constants and functions are already placed in ROM. All symbols placed in RAM must be initialized before the `main` function is called. The linker has already divided the available RAM into different areas for variables, stack, heap, etc.

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

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

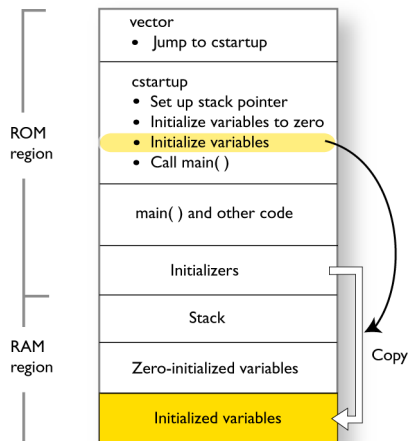


- Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

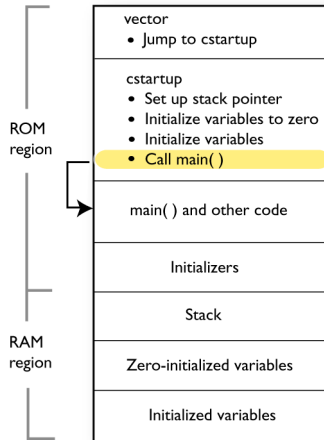


Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`

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



4 Finally, the `main` function is called:



For more information about each stage, see *System startup and termination*, page 131. For more information about initialization of data, see *Initialization at system startup*, page 93.

## THE EXECUTION PHASE

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

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

## THE TERMINATION PHASE

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

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

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

For more information about this, see *System termination*, page 134.

---

## Building applications—an overview

In the command line interface, this line compiles the source file `myfile.c` into the object file `myfile.r82` using the default settings:

```
iccavr32 myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 54.

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

```
xlink myfile.r82 myfile2.r82 -o a.d82 -f my_configfile.xcl -r
```

In this example, `myfile.r82` and `myfile2.r82` are object files, and `my_configfile.xcl` is the linker configuration file. The option `-o` specifies the name of the output file. The option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

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



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

---

## Basic project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler and linker generate the best code for the AVR32 device you are using. You can specify the options either from the command line interface or in the IDE.

You need to make settings for:

- Processor configuration
- Data model
- Code model
- Optimization settings
- Runtime environment

- Customizing the XLINK configuration, see the chapter *Linking your application*.

In addition to these settings, many other options and settings can fine-tune the result even further. For information about how to set options and for a list of all available options, see the chapter *Compiler options* and the *IDE Project Management and Building Guide for AVR32*, respectively.

## PROCESSOR CONFIGURATION

To make the compiler generate optimum code, you should configure it for the AVR32 microprocessor you are using.

### Core

The compiler supports both the avr32a and the avr32b micro architectures. This option has implications for the options used for enabling and disabling the available instruction set extensions, see *Instruction set extensions*, page 56.



Use the `--core` option to select the micro architecture for which the code will be generated.



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

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

### Device

The `--cpu` option is used for declaring the specific *device* that is used, in Atmel's nomenclature referred to as *part*; or for a *family of devices*, referred to as a *platform*. It is not possible to use both `--core` and `--cpu` at the same time.

For a list of supported devices, see the `supported_devices.htm` file available in the `doc` directory.



Use the `--cpu` option to select the device for which the code will be generated.



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

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

## Instruction set extensions

These options can be used for enabling the `fpu`, `simd`, `rmw`, `dsp`, and `flashvault` blocks of instructions:

```
--avr32_fpu_instructions
--avr32_simd_instructions
--avr32_rmw_instructions
--avr32_dsp_instructions
--avr32_flashvault
```

These options can be used together with the `--core` option to control the generated code. By default, the `simd` and `dsp` blocks of instructions are enabled when compiling for the `avr32b` architecture, and the `rmw` block of instructions is enabled when compiling for the `avr32a` architecture.

## Hard and soft alignment

By default, the compiler generates code that adheres to the default alignment for each basic type. Use the `--unaligned_word_access` option to allow the compiler to use the instructions `LD.W` and `ST.W` to access unaligned data, for example when performing parallel path recombination.

For more information about alignment, see *Extensions for embedded systems programming*, page 190.

## DATA MODEL

In the compiler, you can set a default memory access method by selecting a data model. These data models are supported:

- The *small* data model can access the lower and upper 1 Mbyte of memory.
- The *large* data model can access the entire address range.

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

## CODE MODEL

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

- The *small* code model can access the lower 1Mbyte of memory.
- The *medium* code model allows an application no larger than 1 Mbyte to be placed anywhere in memory.
- The *large* code model can access the entire memory.



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

## OPTIMIZATION FOR SPEED AND SIZE

The compiler's optimizer performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, peephole transformations, parallel data path recombination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

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

## RUNTIME ENVIRONMENT

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

To set up an efficient runtime environment you need a good understanding of the various features, see the chapter *The DLIB runtime environment*.



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

The library is automatically chosen according to the settings you make in **Project>Options>General Options**, on the pages **Target**, **Library Configuration**, **Library Options**. A correct include path is automatically set up for the system header files and for the device-specific include files.

Note that for the DLIB library there are different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, etc. See *Library configurations*, page 136, for more information.



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

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

```
d1_libname.r82
```

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

```
-I MyApplication\inc
```

For information about the prebuilt library object files, see *Using prebuilt libraries*, page 121. Make sure to use the object file that matches your other project options.

### Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 123.
- The size of the stack and the heap, see *Setting up stack memory*, page 109, and *Setting up heap memory*, page 109, respectively.

# Data storage

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

---

## Introduction

The AVR32 microprocessor has one continuous main memory space of 4 Gbytes shared between code and data. The direct cost of accessing data is the same regardless of where the data is located. However, the location of the data within the main memory still determines the access cost indirectly, because an address within the first and last megabyte of memory can be handled directly by some instructions.

Different types of physical memory can be placed anywhere 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 AVR32 microprocessor also has two separate sets of registers—the system register file and the debug register file—which can be accessed using special instructions. These registers cannot be accessed through the main memory. The system register file is used for storing information about the microprocessor core whereas the debug registers give access to registers used, for example, by the JTAG debug interface.

**Note:** Some devices have an MMU which uses segment translation. For information about how to use the linker for such a system, see *Linking for segment-translated systems*, page 114.

## DIFFERENT WAYS TO STORE DATA

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

- Auto variables

All variables that are local to a function, except those declared static, are stored either in registers or on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid. For more information, see *Storage of auto variables and parameters*, page 68.

- Global variables, module-static variables, and local variables declared `static`  
In this case, the memory is allocated once and for all. The word `static` in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Data models*, page 66 and *Memory types*, page 60.
- Dynamically allocated data.  
An application can allocate data on the *heap*, where the data remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 69.

---

## Memory types

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 the `data21` memory access method is called memory of `data21` type, or simply `data21` 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 by using dedicated memory attributes. 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.

## Data17

The data17 memory consists of the lower and upper 128 Kbytes of data memory. In hexadecimal notation, this is the address range 0x00000000-0x0001FFFF and 0xFFFE0000-0xFFFFFFFF.

The size of a data17 object is only limited by the allowed memory range, data17 is only available when the RMW instruction set extensions are available, see *--avr32\_rmw\_instructions*, page 244.

For more information, see *\_\_data17*, page 295 and *Memory access methods*, page 180.

## Data21

The data21 memory consists of the lower and upper 1 Mbyte of memory. In hexadecimal notation, this is the address range 0x00000000-0x000FFFFF and 0xFFFF0000-0xFFFFFFFF. To locate a variable explicitly in data21 memory, use the *\_\_data21* memory attribute. Data21 memory is the default memory in the small data model.

The size of a data21 object is only limited by the allowed memory range.

For more information, see *\_\_data21*, page 295 and *Memory access methods*, page 180.

## Data32

Using this memory type, you can place the data objects anywhere in the 4 Gbytes of memory. To locate a variable explicitly in data32 memory, use the *\_\_data32* memory attribute. Data32 memory is the default memory in the large data model.

The *\_\_data32* extended keyword is most useful when you use the small data model, but cannot fit all variables or constants into the first megabyte of memory. You can select some of the variables or constants to be placed above address 0x100000 and declare them with the *\_\_data32* keyword. Because you will mainly benefit from locating variables below address 0x100000 (using the keyword *\_\_data21*) when the variable is accessed directly, variables or constants that are mainly accessed via pointers are especially suited for declaration with the *\_\_data32* keyword.

For more information, see *\_\_data32*, page 296 and *Memory access methods*, page 180.

## System register file

The *\_\_sysreg* keyword is used for declaring variables placed in the system register file. Variables declared in the system register file give you an intuitive way of accessing the parameters that control the execution behavior of the AVR32 microprocessor; for example the location of the exception vector table is controlled by the system register EVBA.

It is not possible to create a pointer type that points at the system register file as there are no instructions that can access the system register file indirectly. Therefore, all variables declared with the `__sysreg` keyword must be located (see *Data placement at an absolute location*, page 215). The AVR32 microprocessor also requires that the variables placed in the system register file are accessed 32 bits at a time, which in turn forces all variables to be aligned to an even word boundary (alignment 4). Also note that the *Atmel AVR32 Architecture Document* sometimes refers to the system registers by using the register number instead of the register *address*. To get the address of a system register from the register number, simply multiply the register number by 4.

Declaring a variable that accesses, for example, the CPU Control Register `CPUCR` register (register number 3 and register address 12) could look like this:

```
__no_init __sysreg unsigned int CPUCR @ 0x00C;
```

The following code can be used for checking whether the return stack feature is available in the current device:

```
if (CPUCR & 0x00000008)
{
    /* Code to run if the return stack is available */
}
```

In practice, you will most likely access the system registers using a predefined include file, so the declaration of system variables is already handled for you. See *Accessing special function registers*, page 227.

## Debug register file

The `__dbgreg` keyword is used for declaring variables placed in the debug register file. Variables declared in the debug register file give you an intuitive way of accessing the registers that control the behavior of the OCD-System (On Chip Debug). Normally, you do not need this keyword, except if you develop system code that should be an interface to external debug hardware like emulators and JTAG probes.

It is not possible to create a pointer type that points at the debug register file as there are no instructions that can access the debug register file indirectly. Therefore, all variables declared with the `__dbgreg` keyword must be located (see *Data placement at an absolute location*, page 215). The AVR32 microprocessor also requires that the variables placed in the system register file are accessed 32 bits at a time, which in turn forces all variables to be aligned to an even word boundary (alignment 4). Also note that the *Atmel AVR32 Architecture Document* sometimes refers to the system registers by using the register *number* instead of the register *address*. To get the address of a system register from the register number, simply multiply the register number by 4.

## USING DATA MEMORY ATTRIBUTES

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

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

Memory type	Keyword	Address range	Pointer size	Default in data model
Data17	<code>__data17</code>	0x0-0x0001FFFF, 0xFFFFE0000-0xFFFFFFFF	32 bits	--
Data21	<code>__data21</code>	0x0-0x000FFFFF, 0xFFF00000-0xFFFFFFFF	32 bits	Small
Data32	<code>__data32</code>	0x0-0xFFFFFFFF	32 bits	Large
System register file	<code>__sysreg</code>	0x0-0x3FC	--	--
Debug register file	<code>__dbreg</code>	0x0-0x3FC	--	--

Table 3: Memory types and their corresponding memory attributes

The keywords are only available if language extensions are enabled in the compiler.



In the IDE, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See `-e`, page 254 for additional information.

For more information about each keyword, see *Descriptions of extended keywords*, page 293.

### Syntax for type attributes used on data objects

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

```
__data17 int i;
int __data17 j;
```

Both `i` and `j` are placed in `data17` memory.

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

```
int __data17 * p;      /* integer in data17 memory */
int * __data17 p;     /* pointer in data17 memory */
__data17 int * p;     /* variable in data17 memory */
```

The integer pointed to by `p1` is in `data17` memory. The variable `p2` is placed in `data17` memory, as is the variable `p3`. In the first two cases, the type attribute behaves in the same way as `const` and `volatile` would.

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

Using a type definition can sometimes make the code clearer:

```
typedef __data17 d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in `data17` 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=__data17
int * q2;
```

The variable `q2` is placed in `data17` memory.

For more examples of using memory attributes, see *More examples*, page 65.



## Type definitions

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

```
/* Defines via a typedef */
typedef char __data21 Byte;
typedef Byte *BytePtr;
Byte aByte;
BytePtr aBytePointer;

/* Defines directly */
__data21 char aByte;
char __data21 *aBytePointer;
```

## STRUCTURES AND MEMORY TYPES

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

In the example below, the variable `gamma` is a structure placed in `data17` memory.

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

__data21 struct MyStruct gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
    int mAlpha;
    __data21 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 `data17` memory is declared. The function returns a pointer to an integer

in data21 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 __data17 myB;</code>	A variable in data17 memory.
<code>__data21 int myC;</code>	A variable in data21 memory.
<code>int * myD;</code>	A pointer stored in default memory. The pointer points to an integer in default 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 197.

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 197.

Because all pointers have data32 representation, there are no restrictions in placement.

---

## Data models

Technically, the data model specifies the default memory type for non-constant objects. This means that the data model controls the placement of static and global variables. The pointer size is the same for all memory types. Dynamically allocated data and data located on the runtime stack are not affected by the data model and can be located anywhere in the 32-bit address space.

**Note:** The placement of `const` declared objects is determined by the selected code model.

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 63.

## SPECIFYING A DATA MODEL

Two data models are implemented: Small, and Large. These models are controlled by the `--data_model` option. Each model has a default memory type and a default pointer size. If you do not specify a data model option, the compiler will use the Small data model for AVR32A devices and the Large data model for AVR32B devices.

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 63.

This table summarizes the different data models:

Data model name	Default memory attribute	Default pointer attribute	Placement of data
Small (default)	<code>__data21</code>	<code>__data32</code>	The lower and upper 1 Mbyte of memory
Large	<code>__data32</code>	<code>__data32</code>	The entire 4 Gbytes of memory

Table 4: Data model characteristics



See the *IDE Project Management and Building Guide for AVR32* 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 249.

### The small data model

The small data model places data in the first 64 Kbytes of memory. This is the only memory that can be accessed using 16-bit pointers. The advantage is that only 16 bits are needed for pointer storage. The default pointer type passed as a parameter will use one register or 2 bytes on the stack.

### The large data model

The large data model places data in the first 16 Mbytes of memory. This is the only memory that can be accessed using 24-bit pointers. The default pointer type passed as a parameter will use one register or 3 bytes on the stack.

---

## Storage of auto variables and parameters

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

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

### THE STACK

The stack can contain:

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

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

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

See also *Stack considerations*, page 205 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. This will happen when one function calls another, which in turn calls a third, etc., and the sum of the stack usage of each function is larger than the size of the stack. The risk is higher if large data objects are stored on the stack, or when recursive functions are used.

---

## Dynamic memory on the heap

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

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

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

See also *Setting up heap memory*, page 109 .

### POTENTIAL PROBLEMS

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

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

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

There is also the matter of fragmentation; this means a heap where small sections of free memory is separated by memory used by allocated objects. It is not possible to allocate

a new object if no piece of free memory is large enough for the object, even though the sum of the sizes of the free memory exceeds the size of the object.

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

# Functions

- Function-related extensions
- Code models and memory attributes for function storage
- Primitives for interrupts, concurrency, and OS-related programming
- Execution in RAM
- Implementing middleware using FlashVault™
- Inlining functions

---

## Function-related extensions

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

- Control 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 211. 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 specify in which part of memory the compiler should place functions by default. Technically, the code models control the following:

- The possible memory range for storing the functions
- The default memory attribute
- The placement of `const` declared variables.

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.

If you do not specify a code model, the compiler will use the Small code model as default for the AVR32B architecture, and the Medium code model for the AVR32A architecture.



See the *IDE Project Management and Building Guide for AVR32* 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 246.

## 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	Default in code model	Description
<code>__code21</code>	0-0x00FFFFFF, 0xFFFF0000-0xFFFFFFFF	4 bytes	Small	Functions can be placed in the lower 1 Mbyte of memory. The upper 1 Mbyte of memory is used for special function registers.
<code>__code32</code>	0-0xFFFFFFFF	4 bytes	Large Medium	Functions can be placed anywhere in the 4 Gbytes of memory.

Table 5: 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 282.

The `__code32` extended keyword is most useful when you use the Small code model, but cannot fit all functions into the first megabyte of memory. You can then select some of the functions to be placed above address 0x100000 and declare them with the `__code32` keyword.

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 AVR32 provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords: `__interrupt`, `__exception`, `__acall`, `__scall`, `__flashvault`, `__flashvault_impl`, `__nested`, `__imported` and `__monitor`
- The pragma directive `#pragma exception`, `#pragma flashvault_vector`, `#pragma handler`, and `#pragma shadow_registers`
- 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 microprocessor 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 AVR32 microprocessor supports many interrupt sources and allows great flexibility with regard to interrupt levels and handlers. One or several interrupt sources are associated with a group and for each group an interrupt routine can be written. Each interrupt routine is associated with a group number and an interrupt level number, which are specified in the AVR32 microprocessor documentation from the chip manufacturer. An interrupt handler can service one or more interrupt groups. For the AVR32 microprocessor, the interrupt handlers must be located within the 16 Mbyte of memory starting from the address in the `EVBA` register.

For more information about the runtime environment used by interrupt routines, see the chapter *Assembler language interface*. See also *shadow\_registers*, page 325, for information about how an interrupt routine can execute without having to save the contents of registers that are used.

#### Interrupt vectors and the interrupt vector table

For the AVR32 microprocessor, the exception table always starts at the address stored in the `EVBA` system register. The exception table contains functions that handle

exceptional processor states, for example unaligned accesses, illegal instructions, and privilege violations. There is no interrupt vector table; instead this information is stored directly in the interrupt controller by the startup code, see *System startup*, page 132.

If a vector is specified in the definition of an interrupt function, an entry is added to the interrupt controller initialization table in the `HTAB` linker segment. This table is parsed during the system startup process, see *System startup*, page 132.

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

### Defining an interrupt function—an example

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

```
#include <avr32\iouc3a0512.h>

#pragma handler = AVR32_EIC_IRQ_GROUP,1/* 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.

### Nested interrupts

When the AVR32 microprocessor handles an interrupt, it immediately disables all other interrupt sources of equal or lower priority to prevent *nesting* of interrupts (the activation of the same interrupt before the current interrupt handler has completed). However, it is sometimes desirable to allow nesting of interrupts, for example when an interrupt handler needs to handle both timing-critical and non-critical processing. An example is when data needs to be fetched from a peripheral unit as quickly as possible but the processing of the data is not timing-critical. The system can then install the handler as a high-priority interrupt and allow nesting. Once the timing-critical part has been executed, the handler can lower the interrupt level to allow other interrupts to activate.

To declare a nestable interrupt handler, add the `__nested` keyword to the interrupt declaration. For example:

```
#pragma handler = 12,3
__nested __interrupt void PORT_Handler()
{
    /* Do timing-critical processing here */
    /* Enable all interrupts by clearing the */
    /* interrupt and general mask bits */
    SR = SR & ~0x1F0000UL;

    /* Do non-critical processing here */
} /* Interrupt masks will automatically be restored here */
```

## Unhandled interrupts

For an application that has more interrupt sources than it actually uses, it is important to make sure that any spurious interrupts from the unused interrupt sources do not cause the system to crash. One way to achieve this is to install one handler for all unhandled interrupts. A default interrupt handler `__unhandled_interrupt` can be found in the `avr32\src\lib` directory.



To use this special interrupt handler in the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Runtime** and select the option **Handle unhandled interrupts**.



To alter the normal startup code and use this special handler from the command line, use the XLINK command line option `--g__init_all_ihandlers`.

## EXCEPTION HANDLERS

If the AVR32 microprocessor encounters a condition that it cannot handle, for example if unaligned data is accessed or an illegal instruction is executed, the processor throws an exception. These exceptions can then be caught by one of the exception handlers installed in the application. This handler should rectify the problem and return to the application that caused the exception or, if the problem cannot be corrected, terminate the execution.

To declare an exception handler, use the `__exception` keyword like this:

```
#pragma exception=0x50,0x10
__exception void ITLB_Miss()
{
    /* Code for handling missing ITLB entry */
}
```

An exception handler can have an optional parameter. In that case, this parameter contains the PC of the instruction that caused the exception. The handler may also return a 32-bit value in which case this value is used as return address. For example:

```
#pragma exception=0x20,4
__exception unsigned long IllegalOpcode(unsigned long LR)
{
    if((* (unsigned short *)LR) &0xE000 == 0xE000)
    {
        return LR + 4;
    }
    else
    {
        return LR + 2;
    }
}
```

Note specifically the handler of the SCALL instruction (used for implementing \_\_scall functions). The runtime library does not support a default implementation of this handler.

The standard startup code will initialize the EVBA register if one or more exception handlers are present in the application.

## Unhandled exceptions

Most applications will not handle any exceptions, or will only handle a small subset of the possible exceptions. In these cases, it is good programming practice to install a default exception handler which will be activated if anything unintended occurs. This handler can then make sure that the application terminates in an orderly way before restarting the system.

A prototype for such a handler can be found in the include file `intrinsics.h`:

```
__exception
void _ _unhandled_exception(unsigned int exception_number,
                             void * offending_PC);
```



To install this default exception handler in the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Runtime** and select the option **Handle unhandled exceptions**.



To use this special exception handler from the command line, use the XLINK command line option `--g__handle_all_exceptions`.

## ACALL FUNCTIONS

The AVR32 microprocessor supports compact function calls via a small special function pointer table. The advantages of these calls, performed by using the `ACALL` instruction, are that:

- Frequently called functions can use a more compact instruction leading to smaller code size
- an application can get access to operating system functions without the application knowing the exact location of those functions.

To make a function use the `ACALL` instruction, declare the function using the `__acall` keyword. The address of the function is stored in the linker segment `ACTAB` and the normal system startup code will automatically initialize the `ACBA` register to point at the `ACTAB` segment.

Normally, the actual location (and thus the vector number) in the `ACTAB` segment is determined at link time and there is no need to explicitly assign a vector number to a function declared with the `__acall` keyword. However, there are times when the exact vector must be specified by using the `#pragma vector` directive before the function declaration.

**Note:** The `ACTAB` segment is bypassed when assigning the address of an `ACALL` function to a function pointer.

### Using ACALL functions to allow in-system upgrades

One reason to specify the exact vector numbers is to allow in-system upgrades of an application. To upgrade a function or a number of functions, the application would only need to alter the table in the `ACTAB` segment.

```
/* Replaceable function */
#pragma vector=0x20
__acall void ReplaceableFunction();
```

### Using ACALL functions to access an API

Another case when the exact vector number should be specified in conjunction with the function declaration is when you use `ACALL` functions for accessing an API which is for example implemented by a third-party vendor, or in a case where the function is present in a masked ROM or in a special boot section of the code memory. In these cases, it is important to also let the compiler and linker know that the function is not a part of your application. This is achieved by using the `__imported` keyword in the function declaration.

For example, an API can look something like this:

```
/* Download API */
#pragma vector=0x40
__acall __imported void InitializeDownload();

#pragma vector=0x41
__acall __imported bool StartDownload(void * address,
                                       int length);

#pragma vector=0x42
__acall __imported bool IsDownloadFinished();
```

## SCALL FUNCTIONS

To protect operating system functions, the AVR32 microprocessor supports an application mode and several supervisor modes. For more information about these modes, refer to the hardware reference manual. Many resources and instructions are limited to the supervisor modes and it might therefore be necessary to allow an application that runs in the non-privileged application mode to temporarily gain access to the privileged supervisor modes. This is achieved by using `SCALL` functions. An `SCALL` function causes the processor to enter the supervisor mode and to start executing the exception handler (see *\_\_exception*, page 296) at offset `0x100`.

**Note:** All function parameters to an `SCALL` function must be passed in registers, which means that `SCALL` functions use a more restricted calling convention than normal functions, see *Calling convention*, page 170.

To declare an `SCALL` function, use the `__scall` keyword in the function definition and declaration. Also, because there is only one exception handler for `SCALL` functions, it is imperative that the handler can deduce which function is being called. Typically, this is achieved by using the first parameter to enumerate the required function, for example:

```
/* Example of an SCALL OS interface */
__scall void * OS_API_AllocMemory(int funcNum,
                                 unsigned int size);
__scall void OS_API_FreeMemory(int funcNum, void * pointer,
                               unsigned int size);

#define OS_AllocMemory(size) OS_API_AllocMemory(0x17, size)
#define OS_FreeMemory(ptr, size) OS_API_FreeMemory (0x18, ptr,
                                                    size)
```

The `SCALL` handler will then use the first parameter to deduce which function is actually requested:

```
#pragma exception=0x100:0
__exception unsigned int scall_handler(int funcNum,
                                       unsigned int p1,
                                       unsigned int p2,
                                       unsigned int p3,
                                       unsigned int p4)
{
    switch(funcNum)
    {
        ...
        case 0x17: return OS_IMPL_AllocMemory(p1);
        case 0x18: return OS_IMPL_FreeMemory(p1, p2);
        ...
    }
}
```

## 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. Note that the individual interrupt masks `I0M-I3M` are not restored, so changes to these bits inside the monitor function will apply after the function has exited.

To define a monitor function, you can use the `__monitor` keyword. For more information, see *\_\_monitor*, page 298.



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.

**Note:** Semaphores can also be implemented using assembler-written routines that use the `XCHG` instruction, or C functions that use the `__exchange_memory` or the `__store_conditional` intrinsic functions.

```

/* 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();
    }
}

```

---

## Execution in RAM

The `__ramfunc` keyword makes a function execute in RAM. In other words it places the function in a segment placed in RAM. The function is copied from ROM to RAM at system startup just like any initialized variable, see *System startup*, page 132 and *System termination*, page 134.

The keyword is specified before the return type:

```
__ramfunc void my_func(void);
```

If a `__ramfunc` declared function tries to access ROM, the compiler issues a warning.

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

String literals and other constants can be avoided by using initialized variables. For example, the following lines:

```

__ramfunc void test()
{
    /* myc: initializer in ROM */
    const int myc[] = { 10, 20 };

    /* string literal in ROM */
    msg("Hello");
}

```

can be rewritten as:

```

__ramfunc void test()
{
    /* myc: initialized by cstartup */
    static int myc[] = { 10, 20 };

    /* hello: initialized by cstartup */
    static char hello[] = "Hello";

    msg(hello);
}

```

---

## Implementing middleware using FlashVault™

FlashVault is a technology that allows firmware to be developed and distributed in such a manner that intellectual property is protected. The protection is implemented on hardware level, thus preventing accidental or intentional read-out of the firmware from the device. For more information about FlashVault and the protection and features it offers, contact Atmel representatives.

A FlashVault application consists of two parts, the secured firmware and the application itself. The firmware developer should write an API that allows the application developer to tap into the functionality of the firmware. To trap into and out of the secured state, special call instructions are used: the `SSCALL` and `RETSS` instructions. There are also a number of FlashVault-specific exception vectors; for information about your specific device, see the documentation from Atmel.

When you design FlashVault-based firmware, you must consider how the application part will interact with the firmware: the API. From the application's point of view, the API is defined through include files that contain function declarations and type definitions. Because there is only a single gateway between the secure mode and the application mode, take care to design the API to allow all interaction through this single link of communication.



Stack parameters cannot be used in the firmware API because the secure mode operation has its own stack pointer. In most cases, five parameter registers will be enough. Passing a pointer to a structure containing additional input or output fields can expand the API to an extremely large number of parameters.

The compiler implements FlashVault support by adding two function type attributes, `__flashvault` and `__flashvault_impl`, and two pragma directives, `#pragma vector` and `#pragma flashvault_vector`. For more information, see `__flashvault`, page 297, `__flashvault_impl`, page 297, `vector`, page 328, and `flashvault_vector`, page 315.

## IMPLEMENTING A SINGLE ENTRY POINT API

To implement a single entry point API, you design one API function that uses command enumerations to select between the various operations. Data can be transferred either using parameters or by placing data in structures and sending it to the function. For example:

```
extern __flashvault int Decrypt_API(int cmd,
                                   void const * input, void * output);
```

When you implement an API this way, all parameters are passed directly to the function. Note that stack parameters cannot be used. If the five available parameter registers are not enough, multiple parameters can be written to a structure and the address to that structure is passed as a pointer parameter:

```
typedef struct {
    int command;
    void const * source_pointer;
    void * destination_pointer;
} command_struct_t;
extern __flashvault int Decrypt_API(
    command_struct_t * parameter_block);
```

If the compiler detects a `__flashvault` declared function definition, it automatically generates a FlashVault exception vector for the `SSCALL` instruction vector. All calls to such functions are made using the `SSCALL` instruction and the function returns using the `RETSS` instruction. It is important to remember that the `SSCALL` instruction cannot be used while in secure state—this means that the API function cannot be used from within the firmware code.

## IMPLEMENTING A MULTIPLE ENTRY POINT API

Another way to implement the API is to let the compiler help with differentiating between the API entry points. Adding a `#pragma vector` directive tells the compiler to place the vector number in `R8` before issuing the `SSCALL` instruction. This allows an `SSCALL` handler to use the vector number to determine which API function implementation to actually call. Once that function returns, the handler uses `RETSS` to return to the caller in non-secure mode. This actual API function implementation should return using a normal `RET` instruction:

```
#pragma vector=0
extern __flashvault unsigned int API_Decode(void const * input,
                                             void * output);

#pragma vector=1
extern __flashvault unsigned int API_Encode(void const * input,
                                             void * output);
```

On the application side, whenever a call to a `__flashvault` declared function with a `#pragma vector` is encountered in the source code, the vector number is automatically loaded into R8 and the function is called using the `SSCALL` instruction.

When you develop the firmware, you should replace the `__flashvault` keyword with the `__flashvault_impl` keyword, allowing the compiler to automatically generate a function entry table in the `FVVEC` segment for all defined functions and pull in a special handler for the `SSCALL` instruction from the runtime library. The default handler can be found in the file `src\lib\FlashVault.s82`, especially note the `__bad_sscall` exception handler function that is called if the vector number is out of range. This example shows how to use the C preprocessor so that a common API file can be used:

```
/* Define the symbol FIRMWARE in the firmware project */
#ifdef FIRMWARE
#define FLASHVAULT __flashvault_impl
#else
#define FLASHVAULT __flashvault
#endif

#pragma vector=0
FLASHVAULT unsigned int Initialize(int mode);

#pragma vector=1
FLASHVAULT unsigned int Encode(unsigned int key,
                               unsigned int data);
```

## LOCKING DOWN THE FIRMWARE AT DOWNLOAD

To enable the FlashVault function in the hardware, you must configure a number of fuse bits in the **Fuse Handler** dialog box; for more information, see the *C-SPY® Debugging Guide for AVR32*. When these bits are correctly set, the device will boot in secure mode. The secured area is protected by two configuration registers, `SSADRR` and `SSADRF`, which control the size of the protected RAM and flash areas:

```
__root const struct {
    unsigned int SSADRR : 16;
    unsigned int SSADRF : 16;
} FlashVaultConfig @ 0x8080004 =
{
    1 /* Assign 1024 bytes of RAM for the secure mode */,
    8 /* Assign 8192 bytes of FLASH for the secure mode */
};
```

See the documentation for your device for addresses and scales of these registers.

---

## 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 317.

- `--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 219.

For more information about the function inlining optimization, see *Function inlining*, page 222.



# Linking overview

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

---

## Linking—an overview

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

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

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

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

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

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

---

## Segments and memory

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

### WHAT IS A SEGMENT?

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

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

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

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

### Segment memory type

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

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

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

---

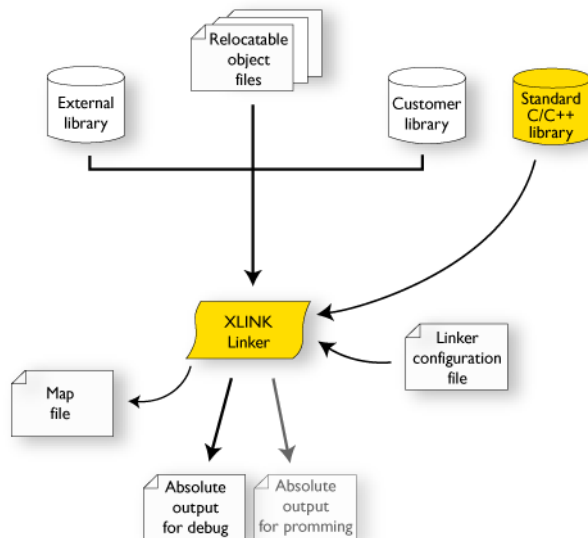
## The linking process in detail

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

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

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

This illustration shows the linking process:



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

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

---

## Placing code and data—the linker configuration file

The placement of segments in memory is performed by the IAR XLINK Linker. It uses a linker configuration file that contains command line options which specify the locations where the segments can be placed, thereby assuring that your application fits on the target microcontroller. To use the same source code with different devices, just rebuild the code with the appropriate linker configuration file.

In particular, the linker configuration file specifies:

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

- The maximum heap size.

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

## THE CONTENTS OF THE LINKER CONFIGURATION FILE

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

- The CPU used:  
`-cavr32`  
 This specifies your target microprocessor.
- Definitions of constants used in the file. These are defined using the XLINK option `-D`. Symbols defined using `-D` can also be accessed from your application.
- The placement directives (the largest part of the linker configuration file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

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

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

For more information about the linker configuration file and how to customize it, see *Linking considerations*, page 105.

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

---

## Initialization at system startup

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

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero
- Variables that are located by use of the `@` operator or the `#pragma location` directive
- Variables that are declared as `const` and therefore can be stored in ROM
- Variables defined with the `__no_init` keyword, meaning that they should not be initialized at all.

## STATIC DATA MEMORY SEGMENTS

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

The names of the segments consist of two parts—the *segment group name* and a *suffix*—for instance, `DATA17_Z`. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for example `DATA17` and `__data17`.

Some of the declared data is placed in non-volatile memory, for example ROM/flash, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more information about segment memory types, see *Segment memory type*, page 90.

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

Categories of declared data	Source	Segment type	Segment name*	Segment content
Zero-initialized data	<code>int i;</code>	Read/write data, zero-init	<code>MEMATTR_Z</code>	None
Zero-initialized data	<code>int i = 0;</code>	Read/write data, zero-init	<code>MEMATTR_Z</code>	None
Initialized data (non-zero)	<code>int i = 6;</code>	Read/write data	<code>MEMATTR_I</code>	None
Initialized data (non-zero)	<code>int i = 6;</code>	Read/write data	<code>MEMATTR_ID</code>	Initializer data for <code>MEMATTR_I</code>
Non-initialized data	<code>__no_init int i;</code>	Read/write data	<code>MEMATTR_N</code>	None
Non-initialized absolute addressed data	<code>__no_init int i @ 0x200;</code>	Read/write data	<code>MEMATTR_AN</code>	None
Constant absolute addressed data	<code>const int i @ 0x200 = 10000;</code>	Read-only data	<code>MEMATTR_AC</code>	The constant
Constants	<code>const int i = 6;</code>	Read-only data	<code>MEMATTR_C</code>	The constant

Table 6: segments holding initialized data

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

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

## THE INITIALIZATION PROCESS

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

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

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

The system startup code initializes the non-zero variables by copying a block of ROM to the location of the variables in RAM. This means that the data in the ROM segment with the suffix `ID` is copied to the corresponding `I` segment.

This works when both segments are placed in continuous memory. However, if one of the segments is divided into smaller pieces, it is very important that:

- The other segment is divided in *exactly* the same way
- It is legal to read and write the memory that represents the gaps in the sequence.
- Segments that contain constants do not need to be initialized; they should only be placed in flash/ROM
- Segments holding `__no_init` declared variables should not be initialized.
- Finally, global C++ object constructors are called.

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

---

## Stack usage analysis

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

In the `AVR32\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 linker configuration file to check that the stack usage calculated by the linker does not exceed the stack space you have allocated.

## PERFORMING A STACK USAGE ANALYSIS

- 1 Enable stack usage analysis:



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



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

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

- 2 Enable the linker map file:



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



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

- 3 Link your project. Note that the linker will issue warnings related to stack usage under certain circumstances, see *Situations where warnings are issued*, page 101.
- 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 97.
- 5 For more details, analyze the call graph log, see *Call graph log*, page 101.

Note that there are limitations and sources of inaccuracy in the analysis, see *Limitations*, page 100.

You might need to specify more information to the linker to get a more representative result. See *Specifying additional stack usage information*, page 99



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





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

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

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

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

When linking, the linker emits an error if the expression is false (zero). In this example, an error will be emitted if the sum of the following exceeds the size of the `MY_STACK` segment:

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

See *check that directive*, page 384.

## RESULT OF AN ANALYSIS—THE MAP FILE CONTENTS

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

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

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

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

Program entry
  "__program_start": 0x0000c0f8

Maximum call chain                               350 bytes

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

interrupt

  "DoStuff()": 0x0000e9ee

Maximum call chain                               4 bytes

  "DoStuff()"                                  4
```

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

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

## SPECIFYING ADDITIONAL STACK USAGE INFORMATION

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

You can:

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

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

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

See also *function directive*, page 385.

- 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 384.

- 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 386 and *calls*, page 308.

- 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 383 and *call\_graph\_root*, page 308.

- 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.r82] to MyFunc13, MyFunc14;
```

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. 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 AVR32*.
- 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.
- C++ source code that uses virtual function calls is not supported.
- If you use other forms of function calls, they will not be reflected in the call graph.
- Using multi-file compilation (`--mfc`) can interfere with using a stack usage control file to specify properties of module-local functions in the involved files.

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



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

## SITUATIONS WHERE WARNINGS ARE ISSUED

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

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

## CALL GRAPH LOG

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

## Example output:

```

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

```

```

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

```

Each line consists of this information:

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





# Linking your application

- Linking considerations
- Linking for segment-translated systems
- Verifying the linked result of code and data placement

---

## Linking considerations



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



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

The `config` directory contains the information required by XLINK, and are ready to be used as is. The only change, if any, you will normally have to make to the supplied configuration file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you must also add details about the external RAM memory area.

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

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

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

## PLACING SEGMENTS

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

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

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

- There is 1 Mbyte addressable memory.
- There is ROM memory in the address ranges 0x0000-0x1FFF, 0x3000-0x4FFF, and 0x10000-0x1FFFF.
- There is RAM memory in the address ranges 0x8000-0xAFFF, 0xD000-0xFFFF, and 0x20000-0x27FFF.
- There are two addressing modes for data, one for data17 memory and one for data21 memory.
- There is one stack and one heap.
- There are two addressing modes for code, one for code21 memory and one for code32 memory.

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

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

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

### General hints for placing segments

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

In addition, you should consider these aspects:

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

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

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

### Using the **-Z** command for sequential placement

Use the `-z` command when you must keep a segment in one consecutive chunk, when you must preserve the order of segment parts in a segment, or, more unlikely, when you must put segments in a specific order.

The following illustrates how to use the `-z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x0000-0x1FFF`.

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

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

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

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

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



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

### Using the **-P** command for packed placement

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

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

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

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

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

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

**Note:** Copy initialization segments—`BASENAME_I` and `BASENAME_ID`—and dynamic initialization segments must be placed using `-Z`.

## PLACING DATA

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

### Placing static memory data segments

Depending on their memory attribute, static data is placed in specific segments. For information about the segments used by the compiler, see *Static data memory segments*, page 94.

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

```
/* First, the segments to be placed in ROM are defined. */
-Z (CONST) DATA17_C=0000-1FFF, 3000-4FFF
-Z (CONST) DATA17_C=0000-1FFF, 3000-4FFF, 10000-1FFFF
-Z (CONST) DATA17_ID, DATA21_ID=010000-1FFFF

/* Then, the RAM data segments are placed in memory. */
-Z (DATA) DATA17_I, DATA16_Z, DATA17_N=8000-AFFF
-Z (DATA) DATA21_I, DATA21_Z, DATA21_N=20000-27FFF
```

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

### Placing located data

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in for example either the `DATA17_AC` or the `DATA17_AN` segment. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker configuration file.

### Placing user-defined segments

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

## SETTING UP STACK MEMORY

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

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

For more information about stack memory, see *Stack considerations*, page 205.



### Stack size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **System** tab.

Add the required stack size in the dedicated text box.



### Stack size allocation from the command line

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

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

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

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

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



### Placing the stack segment

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

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

#### Note:

- This range does not specify the size of the stack; it specifies the range of the available memory.

## SETTING UP HEAP MEMORY

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

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

- The linker segment used for the heap, see *HEAP*, page 378 (DLIB)
- The steps for allocating the heap size, which differs depending on which build interface you are using
- The steps involved for placing the heap segments in memory.

See also *Heap considerations*, page 205.

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



### Heap size allocation in the IDE

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

Add the required heap size in the dedicated text box.



### Heap size allocation from the command line

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

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

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

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

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

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



### Placing the heap segment

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

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

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

## PLACING CODE

This section contains descriptions of the segments used for storing code and the interrupt vector table. For information about all segments, see *Summary of segments*, page 365.

## Startup code

In this example, the segment `CSTART` contains code used during system startup and termination, see *System startup and termination*, page 131. The system startup code should be placed at the location where the chip starts executing code after a reset.

This line will place the `CSTART` segment at the address `0x0100`:

```
-Z (CODE) CSTART=0100-1FFF
```

## Normal code

Depending on their memory attribute, functions are placed in various segments, in this example `CODE21` and `CODE32`.

Placing the segments is a simple operation in the linker configuration file:

```
-P (CODE) CODE21=0000-1FFF,3000-4FFF
-P (CODE) CODE32=0000-1FFF,3000-4FFF,10000-1FFFF
```

Here, the `-P` linker directive is used for allowing `XLINK` to split up the segments and pack their contents more efficiently. This is also useful if the memory range is non-continuous.

For information about segments holding normal code, see the chapter *Segment reference*.

## ACALL jump table

The `ACALL` instructions, supported by the keyword `__acall`, require a table with function addresses. The table is generated automatically by the compiler. For `acall` calls, the instruction will make a direct call to the function using a table index coded into the `ACALL` instructions. See *ACALL functions*, page 77.

The function tables for functions declared `__acall` are stored in a segment named `ACTAB`. In the linker command file, the linker directive for segment placement can look like this:

```
-Z (CONST) ACTAB=0-FFFFFFF
```

## Exception handlers

The exception handlers—functions declared `__exception`—are located in the segments `EVTAB`, `EV100`, and `EVSEG`, where:

- `EVTAB` holds the exception handlers that are only four bytes large, and entries that contain an `RJMP` instruction to the handlers that are larger than four bytes. The `EVBA` register should point at the `EVTAB` segment
- `EV100` holds the scall exception handler at `EVBA + 0x100`

- `EVSEG` holds the exception handlers that are too large to fit the table entries in the `EVTAB` segment.

The exception handler code is located at address `0x80000000` and upwards. By using the special `-Z@` directive, the placement is guaranteed to start from start address and onwards. In the linker command file, the linker directives for segment placement can look like this:

```
-Z@ (CODE) EVTAB=80000000-80FFFFFF
-Z@ (CODE) EV100=80000100-80FFFFFF
-P (CODE) EVSEG=80000000-80FFFFFF
```

For more information, see *Linking for segment-translated systems*, page 114.

### Switch tables

In some cases, a switch statement in C can be implemented as a data structure containing the different case values, and the program locations to jump to for these values. If there are such data structures present, they are placed in a separate (constant data) segment named `SWITCH`. This segment is placed at the desired address in the same way as for other constant segments. The segment may be placed anywhere in main memory, for example:

```
-Z (CONST) SWITCH=0-FFFFFFFE
```

### Interrupt vectors

The interrupt vector table contains pointers to interrupt routines, including the reset routine. In this example, the table is placed in the segment `INTVEC`. The linker directive would then look like this:

```
-Z (CONST) INTVEC=0000-00FF
```

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

### C++ dynamic initialization

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

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

For example:

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

`DIFUNCT` must be placed using `-Z`. For more information, see *DIFUNCT*, page 375.



## KEEPING MODULES

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

```
-A file.r82
```

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

```
-C file.r82
```

## KEEPING SYMBOLS AND SEGMENTS

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

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

For more information about the linking procedure for keeping symbols and sections, see *The linking process in detail*, page 91.

## APPLICATION STARTUP

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

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

## INTERACTION BETWEEN XLINK AND YOUR APPLICATION

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

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

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

## PRODUCING OTHER OUTPUT FORMATS THAN UBROF

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

```
-F intel-standard
```

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

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

---

## Linking for segment-translated systems

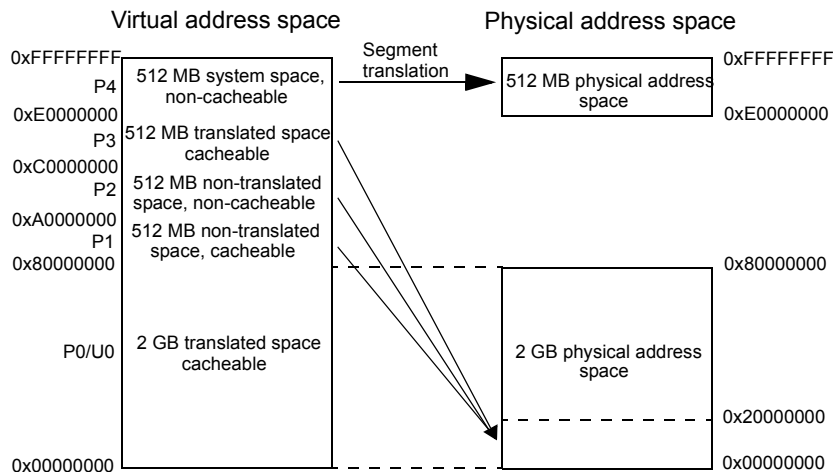
Some of the AVR32 devices have a memory management unit (MMU) which allows an application to define an alternative address space for the application. This means that there are two fundamentally different address spaces: the physical address space and the virtual address space.

The physical address space is related to how the AVR32 microprocessor accesses the physical memories. The physical layout is important because it describes where the actual code and data bytes should be downloaded into the system, and it is also important when creating a ROM image using XLINK. Addresses in the physical address space are called *physical addresses*.

The virtual address space defines how the application, and thus also the debugger, accesses memory. Addresses in the virtual address space are called *virtual addresses*. An application always executes in this address space and the placement restrictions for the different memory types apply to the virtual addresses, see *Using data memory attributes*, page 63. This makes the virtual view very important when choosing code and data models, see *Basic project configuration*, page 54.

## SEGMENT-TRANSLATED MODE

AVR32 devices that feature an MMU usually start executing after reset in a mode called *segment-translated mode*. In this mode, the physical and virtual address spaces differ in three regions:



As you can see in the figure, there are three areas—P1, P2, and P3—in the virtual address space that do not map directly to the physical address space. These three areas are instead mapped by the hardware so that all accesses to the virtual addresses `0x80000000`, `0xA0000000`, or `0xC0000000` are actually performed against the physical address `0x00000000`. This transformation is linear in the entire 512-Mbyte regions, that is `0x80000010` is translated to `0x00000010` and so on.

## IMPLICATIONS FOR THE LINKER CONFIGURATION FILE

The segment translation implies that the linker needs to place all symbols according to the virtual addresses, and store code and data according to the physical addresses. To achieve this, you should for each segment placed in a translated virtual range:

- define an additional segment that will contain the actual bytes
- connect the segments using the `-Q` linker directive.

For example, the segment that contains the code executed at reset is called `RESET` and it must be placed at the virtual address `0xA0000000`; the segment translation process will map this to the physical address `0x00000000`. The additional segment `RESETCODE` is created to contain the actual bytes for the `RESET` segment.

These linker directives are needed:

```
-QRESET=RESETCODE
-Z (CODE) RESET=A0000000-A00003FF
-Z (CODE) RESETCODE=0-3FF
```

The same process must be used for the exception and interrupt handlers located in the three segments EVTAB, EV100, and EVSEG.

```
-QEVTAB=EVBYTES1
-QEV100=EVBYTES2
-QEVSEG=EVBYTES3

-Z@ (CODE) EVTAB=80000400-8007FFFF
-Z@ (CODE) EV100=80000500-8007FFFF
-P (CODE) EVSEG=80000400-8007FFFF

-Z@ (CODE) EVBYTES1=00000400-0007FFFF
-Z@ (CODE) EVBYTES2=00000500-0007FFFF
-P (CODE) EVBYTES3=00000400-0007FFFF
```

Because the virtual addresses are translated, it is imperative that the placement within each translated region—P1, P2, and P3—must exactly match the placement within the destination region (0x00000000–0x1FFFFFFF). A function placed at the virtual address 0x800003C0 must have its code bytes placed at 0x000003C0 or the application will malfunction. To ensure this, all segment-translated linker segments should be linked before any other segments.

## MAPPED MEMORIES

In addition to segment translation, memory in the physical address space can be mapped. A mapped memory range is simply an additional window through which the actual memory can be viewed. Mapped memory is also said to be mirrored. The difference between a mapped memory and the segment translation process is that the memory mirror exists in the physical address space whereas segment translation goes from the virtual address space to the physical address space.

An AVR32 system can contain one or more memory mirrors that are configured either by external hardware or through special configuration bits which are programmed during application download. For more details, refer to the hardware documentation.

To link an application that should run on a system with mapped memory, it is important to choose which view that the application should use. To illustrate this, assume that the memory layout looks like this:

Memory type	Address	Size	Comment
Flash	0x00000000	256 Kbytes	This is the mapped area of the actual memory at address 0x40000000
RAM	0x08000000	32 Kbytes	
Flash	0x40000000	256 Kbytes	

Table 7: Mapped memory, example of

When linking for this system, the linker must be informed that code and data placed in the physical address range 0x00000000–0x0007FFFF should also occupy space in the range 0x40000000–0x4007FFFF. To achieve this, you should add the `-U` linker option to the command line:

```
-U00000000-0007FFFF=04000000-0407FFFF
```

For more information about the `-U` option, see the *IAR Linker and Library Tools Reference Guide*.

**Note:** A system can be segment-translated and mapped at the same time and in that case both linker techniques must be used independently of each other.

## Verifying the linked result of code and data placement

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

### SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

## LINKER MAP FILE

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

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

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

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

For more information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IDE Project Management and Building Guide for AVR32*.

# The DLIB runtime environment

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

---

## Introduction to the runtime environment

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

### RUNTIME ENVIRONMENT FUNCTIONALITY

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

The runtime library is delivered both as prebuilt libraries and (depending on your product package) as source files, and you can find them in the product subdirectories `avr32\lib` and `avr32\src\lib`, respectively.

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

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
  - Peripheral unit registers and interrupt definitions in include files
  - Target-specific `dsp` instructions
  - FlashVault.
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- A floating-point environment (*fenv*) that contains floating-point arithmetics support, see *fenv.h*, page 361.
- Special compiler support, for instance functions for switch handling or integer arithmetics.

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

## SETTING UP THE RUNTIME ENVIRONMENT

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

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

- Choose which runtime library object file to use
 

The IDE will automatically choose a runtime library based on your project settings. If you build from the command line, you must specify the object file explicitly. See *Using prebuilt libraries*, page 121.
- Choose which predefined runtime library configuration to use—Normal or Full
 

You can configure the level of support for certain library functionality, for example, locale, file descriptors, and multibyte characters. If you do not specify anything, a default library configuration file that matches the library object file is automatically used. To specify a library configuration explicitly, use the `--dlib_config` compiler option. See *Library configurations*, page 136.
- Optimize the size of the runtime library
 

You can specify the formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 123.

You can also specify stack and heap size and placement, see *Setting up stack memory*, page 109, and *Setting up heap memory*, page 109, respectively.
- Include debug support for runtime and I/O debugging
 

The library offers support for mechanisms like redirecting standard input and output to the C-SPY **Terminal I/O** window and accessing files on the host computer, see *Application debug support*, page 125.
- Adapt the library for target hardware
 

The library uses a set of low-level functions for handling accesses to your target system. To make these accesses work, you must implement your own version of these functions. For example, to make `printf` write to an LCD display on your board, you must implement a target-adapted version of the low-level function `__write`, so that it can write characters to the display. To customize such functions, you need a good understanding of the library low-level interface, see *Adapting the library for target hardware*, page 128.



- **Override library modules**  
If you have customized the library functionality, you need to make sure your versions of the library modules are used instead of the default modules. This can be done without rebuilding the entire library, see *Overriding library modules*, page 129.
- **Customize system initialization**  
It is likely that you need to customize the source code for system initialization, for example, your application might need to initialize memory-mapped special function registers, or omit the default initialization of data segments. You do this by customizing the routine `__low_level_init`, which is executed before the data segments are initialized. See *System startup and termination*, page 131 and *Customizing system initialization*, page 135.
- **Configure your own library configuration files**  
In addition to the prebuilt library configurations, you can make your own library configuration, but that requires that you *rebuild* the library. This gives you full control of the runtime environment. See *Building and using a customized library*, page 130.
- **Manage a multithreaded environment**  
In a multithreaded environment, you must adapt the runtime library to treat all library objects according to whether they are global or local to a thread. See *Managing a multithreaded environment*, page 149.
- **Check module consistency**  
You can use runtime model attributes to ensure that modules are built using compatible settings, see *Checking module consistency*, page 154.

---

## Using prebuilt libraries

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

- Core
- Code model
- Data model
- Unaligned access
- Floating-point implementation
- Library configuration—Normal or Full.

## CHOOSING A LIBRARY



The IDE will include the correct library object file and library configuration file based on the options you select. See the *IDE Project Management and Building Guide for AVR32* for more information.



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

- Specify which library object file to use on the XLINK command line, like:

```
dllibname.r82
```

- If you do not specify a library configuration, the default will be used. However, you can specify the library configuration explicitly for the compiler:

```
--dlib_config C:\...\dllibname.h
```

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

You can find the library object files and the library configuration files in the subdirectory `avr32\lib\`.

## LIBRARY FILENAME SYNTAX

The names of the libraries are constructed from these elements:

<code>{library}</code>	is <code>dl</code> for the IAR DLIB runtime environment
<code>{core}</code>	is one of <code>avr32a</code> or <code>avr32b</code>
<code>{code_model}</code>	is one of <code>s</code> , <code>m</code> , or <code>l</code> for small, medium, and large code model, respectively
<code>{data_model}</code>	is one of <code>s</code> or <code>l</code> for small and large data model, respectively
<code>{unaligned_access}</code>	is one of <code>a</code> or <code>u</code> for aligned and unaligned access, respectively
<code>{float_impl}</code>	is one of <code>s</code> or <code>h</code> for software implementation and hardware implementation (FPU)
<code>{lib_config}</code>	is one of <code>n</code> or <code>f</code> for Normal and Full, respectively.

For example, the library `dlavr32alsasn.r82` is configured for the large code model, the small data model, aligned access, software implementation, and for the normal library configuration.

## CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

These items can be customized:

Items that can be customized	Described in
Formatters for <code>printf</code> and <code>scanf</code>	<i>Choosing formatters for <code>printf</code> and <code>scanf</code></i> , page 123
Startup and termination code	<i>System startup and termination</i> , page 131
Low-level input and output	<i>Standard streams for input and output</i> , page 137
File input and output	<i>File input and output</i> , page 141
Low-level environment functions	<i>Environment interaction</i> , page 144
Low-level signal functions	<i>Signal and raise</i> , page 145
Low-level time functions	<i>Time</i> , page 145
Some library math functions	<i>Math functions</i> , page 146
Size of heaps, stacks, and segments	<i>Linking your application</i> , page 105

Table 8: Customizable items

For information about how to override library modules, see *Overriding library modules*, page 129.

---

## Choosing formatters for `printf` and `scanf`

The linker automatically chooses an appropriate formatter for `printf`- and `scanf`-related function based on information from the compiler. If that information is missing or insufficient, for example if `printf` is used through a function pointer, if the object file is old, etc, then the automatic choice is the Full formatter. In this case you might want to choose a formatter manually.

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

**Note:** If you rebuild the library, you can optimize these functions even further, see *Configuration symbols for `printf` and `scanf`*, page 139.

### CHOOSING A PRINTF FORMATTER

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 in the Standard C/EC++ library.

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

Table 9: Formatters for printf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 139.



### Manually specifying the print formatter in the IDE

To specify a formatter manually, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



### Manually specifying the printf formatter from the command line

To specify a formatter manually, add one of these lines in the linker configuration file you are using:

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

## CHOOSING A SCANF FORMATTER

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 in the Standard C/C++ library.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	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
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</code> <code>long</code> support	No	No	Yes

Table 10: Formatters for `scanf`

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for `printf` and `scanf`*, page 139.



### Manually specifying the `scanf` formatter in the IDE

To specify a formatter manually, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



### Manually specifying the `scanf` formatter from the command line

To specify a formatter manually, add one of these lines in the linker configuration file you are using:

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

## Application debug support

In addition to the tools that generate debug information, there is a debug version of the library low-level interface (typically, I/O handling and basic runtime support). Using the debug library, your application can perform things like opening a file on the host computer and redirecting `stdout` to the C-SPY **Terminal I/O** window.

## INCLUDING C-SPY DEBUGGING SUPPORT

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

This table describes the different levels of debugging support:

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

*Table 11: Levels of debugging support in runtime libraries*

\* If you build your application project with this level of debugging support, certain functions in the library are replaced by functions that communicate with C-SPY. For more information, see *The debug library functionality*, page 126.

In the IDE, choose **Project>Options>Linker**. On the **Output** page, select the appropriate **Format** option.

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

## THE DEBUG LIBRARY FUNCTIONALITY

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

These capabilities can be valuable during the early development of an application, for example in an application that uses file I/O before any flash file system I/O drivers are implemented. Or, if you need to debug constructions in your application that use `stdin`

and `stdout` without the actual hardware device for input and output being available. Another use is producing debug printouts.

The mechanism used for implementing this feature works as follows:

The debugger will detect the presence of the function `__DebugBreak`, which will be part of the application if you linked it with the `XLINK` option for C-SPY debugging support. In this case, the debugger will automatically set a breakpoint at the `__DebugBreak` function. When the application calls, for example, `open`, the `__DebugBreak` function is called, which will cause the application to break and perform the necessary services. The execution will then resume.

## THE C-SPY TERMINAL I/O WINDOW

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

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

For more information about the **Terminal I/O** window, see the *C-SPY® Debugging Guide for AVR32*.

### Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

## LOW-LEVEL FUNCTIONS IN THE DEBUG LIBRARY

The debug library contains implementations of the following low-level functions:

Function in DLIB low-level interface	Response by C-SPY
<code>abort</code>	Notifies that the application has called <code>abort</code> *
<code>clock</code>	Returns the clock on the host computer
<code>__close</code>	Closes the associated host file on the host computer
<code>__exit</code>	Notifies that the end of the application was reached *
<code>__lseek</code>	Searches in the associated host file on the host computer
<code>__open</code>	Opens a file on the host computer
<code>__read</code>	Directs <code>stdin</code> , <code>stdout</code> , and <code>stderr</code> to the <b>Terminal I/O</b> window. All other files will read the associated host file
<code>remove</code>	Writes a message to the <b>Debug Log</b> window and returns <code>-1</code>
<code>rename</code>	Writes a message to the <b>Debug Log</b> window and returns <code>-1</code>
<code>_ReportAssert</code>	Handles failed asserts *
<code>system</code>	Writes a message to the <b>Debug Log</b> window and returns <code>-1</code>
<code>time</code>	Returns the time on the host computer
<code>__write</code>	Directs <code>stdin</code> , <code>stdout</code> , and <code>stderr</code> to the <b>Terminal I/O</b> window. All other files will write to the associated host file

Table 12: Functions with special meanings when linked with debug library

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

**Note:** You should not use the low-level interface functions prefixed with `_` or `__` directly in your application. Instead you should use the high-level functions that use these functions. For more information, see *Library low-level interface*, page 129.

## Adapting the library for target hardware

The library uses a set of low-level functions for handling accesses to your target system. To make these accesses work, you must implement your own version of these functions. These low-level functions are referred to as the *library low-level interface*.

When you have implemented your low-level interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules*, page 129.



## LIBRARY LOW-LEVEL INTERFACE

The library uses a set of low-level functions to communicate with the target system. For example, `printf` and all other standard output functions use the low-level function `__write` to send the actual characters to an output device. Most of the low-level functions, like `__write`, have no implementation. Instead, you must implement them yourself to match your hardware.

However, the library contains a debug version of the library low-level interface, where the low-level functions are implemented so that they interact with the host computer via the debugger, instead of with the target hardware. If you use the debug library, your application can perform tasks like writing to the **Terminal I/O** window, accessing files on the host computer, getting the time from the host computer, etc. For more information, see *The debug library functionality*, page 126.

Note that your application should not use the low-level functions directly. Instead you should use the corresponding standard library function. For example, to write to `stdout`, you should use standard library functions like `printf` or `puts`, instead of `__write`.

The library files that you can override with your own versions are located in the `avr32\src\lib` directory.

The low-level interface is further described in these sections:

- *Standard streams for input and output*, page 137
- *File input and output*, page 141
- *Signal and raise*, page 145
- *Time*, page 145
- *Assert*, page 148.

---

## Overriding library modules

To use a library low-level interface that you have implemented, add it to your application. See *Adapting the library for target hardware*, page 128. Or, you might want to override a default library routine with your customized version. In both cases, follow this procedure:

- 1 Use a template source file—a library source file or another template—and copy it to your project directory.
- 2 Modify the file.
- 3 Add the customized file to your project, like any other source file.

**Note:** If you have implemented a library low-level interface and added it to a project that you have built with debug support, your low-level functions will be used and not the C-SPY debug support modules. For example, if you replace the debug support module `__write` with your own version, the C-SPY **Terminal I/O** window will not be supported.

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 duplicate definition errors.

The library files that you can override with your own versions are located in the `avr32\src\lib` directory.

---

## Building and using a customized library

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

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

In those cases, you must:

- Set up a library project
- Make the required library modifications
- Build your customized library
- Finally, make sure your application project will use the customized library.

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

For information about the build process and the IAR Command Line Build Utility, see the *IDE Project Management and Building Guide for AVR32*.

### SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template uses the Full library configuration, see Table 13, *Library configurations*.



In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 54.

**Note:** There is one important restriction on setting options. If you set an option on file level (file level override), no options on higher levels that operate on files will affect that file.

## MODIFYING THE LIBRARY FUNCTIONALITY

You must modify the library configuration file and build your own library if you want to modify support for, for example, locale, file descriptors, and multibyte characters. This will include or exclude certain parts of the runtime environment.

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLIB_Defaults.h`. This read-only file describes the configuration possibilities. Your library also has its own library configuration file `dlavr32Custom.h`, which sets up that specific library with the required library configuration. For more information, see *Customizing a prebuilt library without rebuilding*, page 122.

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

### Modifying the library configuration file

In your library project, open the file `dlavr32Custom.h` and customize it by setting the values of the configuration symbols according to the application requirements.

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

## USING A CUSTOMIZED LIBRARY

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

In the IDE you must do these steps:

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

---

## System startup and termination

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

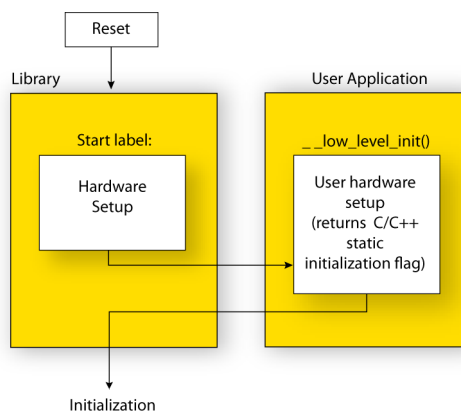
The code for handling startup and termination is located in the source files `cstartup.s82`, `cmain.s82`, `cexit.s82`, and `low_level_init.c` or `low_level_init.s82` located in the `avr32\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 135.

## SYSTEM STARTUP

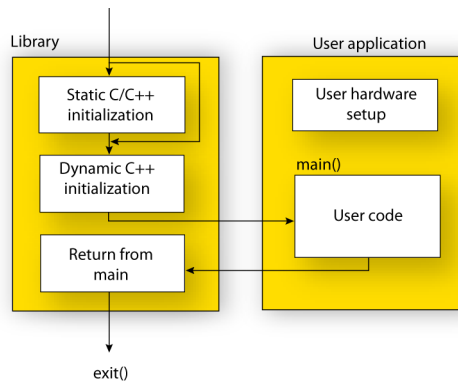
During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

For the hardware initialization, it looks like this:



- When the CPU is reset it will jump to the device-specific reset location—the program entry label `__program_start` in the system startup code.
- The supervisor stack pointer (R13) is initialized to the end of the `SSTACK` (supervisor stack) segment.
- If any `ACALL` functions are called in the application, the Application Call Base Address register (`ACBA`) is initialized to the start address of the `ACTAB` segment.
- If the application contains interrupt or exception handlers, the Exception Vector Base Address register (`EVBA`) is initialized to the start address of the `EVSEG` segment (the exception handler segment).
- If the application contains interrupt handlers, the `__init_ihandlers` function is called, which initializes the interrupt controller with information located in the `HTAB` segment.
- 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 93
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

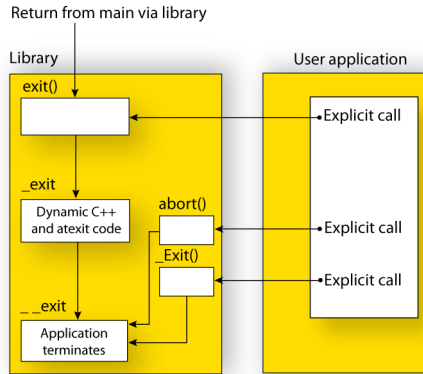
### Stack initialization

The `cstartup` module initializes the supervisor stack pointer to the end of the supervisor stack segment, called `SSTACK`.

The application stack pointer is not by default initialized to point to the application stack segment `CSTACK`. In an application that uses both the supervisor stack and the application stack, the application stack should normally be set up in a special way. Use the `LDMTS` instruction to set up the application stack pointer.

## 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 perform these operations:

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

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

If you want your application to do anything extra at exit, for example resetting the system, you can write your own implementation of the `__exit(int)` function.

### C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *Application debug support*, page 125.

---

## Customizing system initialization

It is likely that you need to customize the code for system initialization. For example, your application might need to initialize memory-mapped special function registers (SFRs), or omit the default initialization of data segments performed by `cstartup`.

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cmain.s` before the data segments are initialized. Modifying the file `cstartup.s82` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s82` and `low_level_init.c`, located in the `avr32\src\lib` directory.

**Note:** Normally, you do not need to customize either of the files `cmain.s82` or `cexit.s82`.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 130.

**Note:** Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s82`, you do not have to rebuild the library.

### `__LOW_LEVEL_INIT`

Two skeleton low-level initialization files are supplied with the product: a C source file, `low_level_init.c` and an alternative assembler source file, `low_level_init.s82`. The latter is part of the prebuilt runtime environment. The only limitation using the C source version is that static initialized variables cannot be used within the file, as variable initialization has not been performed at this point.

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

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

## MODIFYING THE FILE CSTARTUP.S82

As noted earlier, you should not modify the file `cstartup.s82` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s82`, 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 129.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s82`. For information about how to change the start label used by the linker, read about the `-s` option in the *IAR Linker and Library Tools Reference Guide*.

---

## Library configurations

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

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

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

These predefined library configurations are available:

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

Table 13: Library configurations

## CHOOSING A RUNTIME CONFIGURATION

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

- Default prebuilt configuration—if you do not specify a library configuration explicitly you will get the default configuration. A configuration file that matches the runtime library object file will automatically be used.



- Prebuilt configuration of your choice—to specify a runtime configuration explicitly, use the `--dlib_config` compiler option. See *--dlib\_config*, page 253.
- Your own configuration—you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For more information, see *Building and using a customized library*, page 130.

The prebuilt libraries are based on the default configurations.

---

## Standard streams for input and output

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

There are low-level I/O functions, which are the fundamental functions through which C and C++ performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these functions using whatever facilities the hardware environment provides. For more information about implementing low-level functions, see *Adapting the library for target hardware*, page 128.

### IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `avr32\src\lib` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 130. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

**Note:** If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *Application debug support*, page 125.

**Example of using `__write`**

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 `0xFFFF0008`:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 0xFFFF0008;

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;
}
```

**Note:** When DLIB calls `__write`, DLIB assumes the following interface: a call to `__write` where `buf` has the value `NULL` is a command to flush the stream. When the handle is `-1`, all streams should be flushed.

### Example of using `__read`

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at `0xFFFF0010`:

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0xFFFF0010;

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

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

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

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

    return nChars;
}
```

For information about the `@` operator, see *Controlling data and function placement in memory*, page 214.

---

## Configuration symbols for `printf` and `scanf`

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Choosing formatters for `printf` and `scanf`*, page 123.

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

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

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

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

Table 14: Descriptions of printf configuration symbols

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

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

Table 15: Descriptions of scanf configuration symbols

## CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must:

- 1 Set up a library project, see *Building and using a customized library*, page 130.
- 2 Define the configuration symbols according to your application requirements.

## File input and output

The library contains a large number of powerful functions for file I/O operations, 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. Before your application can use the library functions for file I/O operations, you must implement the corresponding low-level function to suit your target hardware. For more information, see *Adapting the library for target hardware*, page 128.

Note that file I/O capability in the library is only supported by libraries with the full library configuration, see *Library configurations*, page 136. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

Template code for these I/O files is included in the product:

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

Table 16: Low-level I/O files

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

**Note:** If you link your application with I/O debug support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Application debug support*, page 125.

## Locale

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

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

The DLIB library can be used in two main modes:

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

## LOCALE SUPPORT IN PREBUILT LIBRARIES

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

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

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

## CUSTOMIZING THE LOCALE SUPPORT

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

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

### Locale configuration symbols

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

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

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 130.

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

### Building a library without support for locale interface

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

### Building a library with support for locale interface

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

## CHANGING LOCALES AT RUNTIME

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

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

*lang\_REGION*

or

*lang\_REGION.encoding*

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

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

### Example

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

```
setlocale (LC_ALL, "sv_FI.UTF8");
```

---

## Environment interaction

According to the C standard, your application can interact with the environment using the functions `getenv` and `system`.

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

### THE GETENV FUNCTION

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `avr32\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 129.

### THE SYSTEM FUNCTION

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

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 130.



**Note:** If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For more information, see *Application debug support*, page 125.

---

## Signal and raise

Default implementations of the functions `signal` and `raise` are available. If these functions do not provide the functionality that you need, you can implement your own versions.

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `avr32\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 129.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 130.

---

## Time

To make the `__time32`, `__time64`, 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 362.

To implement these functions does not require that you rebuild the library. You can find source templates in the files `clock.c`, `time.c`, `time64.c`, and `getzone.c` in the `avr32\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 129.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 130.

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

**Note:** If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For more information, see *Application debug support*, page 125.

---

## Strtod

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

- 1 Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.
- 2 Rebuild the library, see *Building and using a customized library*, page 130.

---

## Math functions

Some library math functions are also available in size-optimized versions, and in more accurate versions.

### SMALLER VERSIONS

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

The names of the functions are constructed like:

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

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

To use these functions, the default function names must be redirected to these names when linking, using the following options:

```
-e__iar_sin_small=sin  
-e__iar_cos_small=cos  
-e__iar_tan_small=tan  
-e__iar_log_small=log  
-e__iar_log2_small=log2  
-e__iar_log10_small=log10  
-e__iar_exp_small=exp  
-e__iar_pow_small=pow  
-e__iar_Sin_small=__iar_Sin  
-e__iar_Log_small=__iar_Log
```

```

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

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

```

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

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

## MORE ACCURATE VERSIONS

The functions `cos`, `pow`, `sin`, and `tan`, and the help functions `__iar_Sin` and `__iar_Pow` exist in versions in the library that are more exact and can handle larger argument ranges. The drawback is that they are larger and slower than the default versions.

The names of the functions are constructed like:

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

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

To use these functions, the default function names must be redirected to these names when linking, using the following options:

```
-e__iar_sin_accurate=sin
-e__iar_cos_accurate=cos
-e__iar_tan_accurate=tan
-e__iar_pow_accurate=pow
-e__iar_Sin_accurate=__iar_Sin
-e__iar_Pow_accurate=__iar_Pow

-e__iar_sin_accuratef=sinf
-e__iar_cos_accuratef=cosf
-e__iar_tan_accuratef=tanf
-e__iar_pow_accuratef=powf
-e__iar_Sin_accuratef=__iar_Sinf
-e__iar_Pow_accuratef=__iar_Powf

-e__iar_sin_accuratel=sinl
-e__iar_cos_accuratel=cosl
-e__iar_tan_accuratel=tanl
-e__iar_pow_accuratel=powl
-e__iar_Sin_accuratel=__iar_Sinl
-e__iar_Pow_accuratel=__iar_Powl
```

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

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

---

## Assert

If you linked your application with the option **With I/O emulation modules**, C-SPY will be notified about failed asserts. If this is not the behavior you require, you can add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `avr32\src\lib` directory. For more information, see *Overriding library modules*, page 129. To turn off assertions, you must define the symbol `NDEBUG`.



In the IDE, this symbol `NDEBUG` is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See *NDEBUG*, page 353.

## Managing a multithreaded environment

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

To configure a customized library with multithread support, add the line `#define _DLIB_THREAD_SUPPORT 3` in the library configuration file and rebuild your library.

The low-level implementations of locks and TLS are system-specific, and is not included in the DLIB library. 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 LIBRARY

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

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

These library objects use TLS:

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

Table 17: Library objects using TLS

Library objects using TLS	When these functions are used
Miscellaneous functions	atexit, strtok

Table 17: Library objects using TLS (Continued)

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

If one of the C++ variants is used together with a DLIB library 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 the runtime environment on the command line, for use with threaded applications, use the linker option `--threaded_lib`.



To configure the runtime environment in the IDE for use with threaded applications, choose **Project>Options>Linker>Extra Options** and specify the linker option `--threaded_lib`. If you are using C++, you must also choose

**Project>Options>C/C++ Compiler>Extra Options** and specify the compiler option `--guard_calls`.

To complement the built-in multithreaded support in the 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 or redirect the interface to the system locks interface (using the linker option `-e`)
- Implement code that handles thread creation, thread destruction, and TLS access methods for the library
- Modify the linker configuration file accordingly.

You can find the required declaration of functions and definitions of macros in the `DLib_Threads.h` file, which is included by `yvals.h`.

**Note:** If you are using a third-party RTOS, check their guidelines for how to enable multithread support with IAR Systems tools.

## System locks interface

This interface must be fully implemented for system locks to work:

```
typedef void *__iar_Rmtx;                /* Lock info object */

void __iar_system_Mtxinit(__iar_Rmtx *); /* Initialize a system
                                          lock */
void __iar_system_Mtxdst(__iar_Rmtx *); /* Destroy a system lock */
void __iar_system_Mtxlock(__iar_Rmtx *); /* Lock a system lock */
void __iar_system_Mtxunlock(__iar_Rmtx *); /* Unlock a system
                                           lock */
```

The lock and unlock implementation must survive nested calls.

## File streams locks interface

This interface is only needed for the Full library configuration. If file streams are used, they can either be fully implemented or they can be redirected to the system locks interface. This interface must be implemented for file streams locks to work:

```
typedef void *__iar_Rmtx;                /* Lock info object */

void __iar_file_Mtxinit(__iar_Rmtx *); /* Initialize a file lock */
void __iar_file_Mtxdst(__iar_Rmtx *); /* Destroy a file lock */
void __iar_file_Mtxlock(__iar_Rmtx *); /* Lock a file lock */
void __iar_file_Mtxunlock(__iar_Rmtx *); /* Unlock a file lock */
```

The lock and unlock implementation must survive nested calls.

## DLIB lock usage

The number of locks that the DLIB library assumes exist are:

- `_FOPEN_MAX`—the maximum number of file stream locks. These locks are only used in the Full library configuration, in other words only if both the macro symbols `_DLIB_FILE_DESCRIPTOR` and `_FILE_OP_LOCKS` are true.
- `_MAX_LOCK`—the maximum number of system locks.

Note that even if the application uses fewer locks, the DLIB library will initialize and destroy all of the locks above.

For information about the initialization and destruction code, see `xsyslock.c`.

## TLS handling

The DLIB library supports TLS memory areas for two types of threads: the *main thread* (the `main` function including the system startup and exit code) and *secondary threads*.

The main thread's TLS memory area:

- Is automatically created and initialized by your application's startup sequence
- Is automatically destructed by the application's destruct sequence
- Is located in the segment `__DLIB_PERTHREAD`
- Exists also for non-threaded applications.

Each secondary thread's TLS memory area:

- Must be manually created and initialized
- Must be manually destructed
- Is located in a manually allocated memory area.

If you need the runtime library to support secondary threads, you must override the function:

```
void *__iar_dlib_perthread_access(void *symp);
```

The parameter is the address to the TLS variable to be accessed—in the main thread's TLS area—and it should return the address to the symbol in the current TLS area.

Two interfaces can be used for creating and destroying secondary threads. You can use the following interface that allocates a memory area on the heap and initializes it. At deallocation, it destroys the objects in the area and then frees the memory.

```
void *__iar_dlib_perthread_allocate(void);
void __iar_dlib_perthread_deallocate(void *);
```

Alternatively, if the application handles the TLS allocation, you can use this interface for initializing and destroying the objects in the memory area:

```
void __iar_dlib_perthread_initialize(void *);
void __iar_dlib_perthread_destroy(void *);
```

These macros can be helpful when you implement an interface for creating and destroying secondary threads:

Macro	Description
<code>__IAR_DLIB_PERTHREAD_SIZE</code>	The size needed for the TLS memory area.
<code>__IAR_DLIB_PERTHREAD_INIT_SIZE</code>	The initializer size for the TLS memory area. You should initialize the rest of the TLS memory area, up to <code>__IAR_DLIB_PERTHREAD_SIZE</code> to zero.
<code>__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(symbolptr)</code>	The offset to the symbol in the TLS memory area.

*Table 18: Macros for implementing TLS allocation*



Note that the size needed for TLS variables depends on which DLIB resources your application uses.

This is an example of how you can handle threads:

```
#include <yvals.h>

/* A thread's TLS pointer */
void _DLIB_TLS_MEMORY *TLSp;

/* Are we in a secondary thread? */
int InSecondaryThread = 0;

/* Allocate a thread-local TLS memory
   area and store a pointer to it in TLSp. */
void AllocateTLS()
{
    TLSp = __iar_dlib_perthread_allocate();
}

/* Deallocate the thread-local TLS memory area. */
void DeallocateTLS()
{
    __iar_dlib_perthread_deallocate(TLSp);
}

/* Access an object in the
   thread-local TLS memory area. */
void _DLIB_TLS_MEMORY *__iar_dlib_perthread_access(
    void _DLIB_TLS_MEMORY *sympb)
{
    char _DLIB_TLS_MEMORY *p = 0;
    if (InSecondaryThread)
        p = (char _DLIB_TLS_MEMORY *) TLSp;
    else
        p = (char _DLIB_TLS_MEMORY *)
            __segment_begin("__DLIB_PERTHREAD");

    p += __IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(sympb);
    return (void _DLIB_TLS_MEMORY *) p;
}
```

The `TLSp` variable is unique for each thread, and must be exchanged by the RTOS or manually whenever a thread switch occurs.

## TLS IN THE LINKER CONFIGURATION FILE

If threads are used, the main thread's TLS memory area must be initialized by plain copying because the initializers are used for each secondary thread's TLS memory area as well. This is controlled by the following statement in your linker configuration file:

```
-Q__DLIB_PERTHREAD=__DLIB_PERTHREAD_init
```

Both the `DLIB_PERTHREAD` segment and the `__DLIB_PERTHREAD_init` segment must be placed in default memory for RAM and ROM, respectively.

The startup code will copy `__DLIB_PERTHREAD_init` to `DLIB_PERTHREAD`.

---

## Checking module consistency

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

For example, in the compiler, it is possible to specify for which core the code should be generated. If you write a routine that only works for the AVR32A core, it is possible to check that the routine is not used in an application built for the AVR32B core.

### RUNTIME MODEL ATTRIBUTES

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

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

### Example

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

Object file	Color	Taste
file1	blue	not defined
file2	red	not defined

*Table 19: Example of runtime model attributes*

Object file	Color	Taste
file3	red	*
file4	red	spicy
file5	red	lean

Table 19: Example of runtime model attributes (Continued)

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

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

## USING RUNTIME MODEL ATTRIBUTES

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

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

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

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

Note that key names that start with two underscores are reserved by the compiler. For more information about the syntax, see *rtmodel*, page 323 and the *IAR Assembler User Guide for AVR32*, respectively.

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

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

## PREDEFINED RUNTIME ATTRIBUTES

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

Runtime model attribute	Value	Description
<code>__rt_version</code>		This runtime key is always present in all modules generated by the compiler. If a major change in the runtime characteristics occurs, the value of this key changes.
<code>__code_model</code>	small, medium, or large	Reflects the <code>--code_model</code> option used in the project.
<code>__core</code>	avr32a or avr32b	Reflects the <code>--core</code> option in use.
<code>__data_model</code>	small or large	Reflects the <code>--data_model</code> option used in the project.
<code>__enum_size</code>	variable or fixed	Reflects the <code>--variable_enum_size</code> option in use.
<code>__unaligned_word_access</code>	enabled or disabled	Reflects the <code>--unaligned_word_access</code> option in use.

Table 20: Predefined runtime model attributes

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

If you are using assembler routines in the C or C++ code, see the chapter *Assembler directives* in the *IAR Assembler User Guide for AVR32*.

### Example

The following assembler source code provides a function that counts the number of times it has been called by increasing the register `R4`. The routine assumes that the application does not use `R4` for anything else, that is, the register has been locked for usage. To ensure this, a runtime module attribute, `__reg_r4`, has been defined with a value `counter`. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not

set this attribute. Note that the compiler sets this attribute to *free*, unless the register is locked.

```

        module          myCounter
        public          myCounter
        section         CODE:CODE
        rtmodel         "__reg_r4", "counter"
myCounter: sub         r4, -1
        ret             r12
        end
```

If this module is used in an application that contains modules where the register R4 has not been locked, an error message is issued by the linker:

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



# 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 AVR32 provides several ways to access low-level resources:

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

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

### INTRINSIC FUNCTIONS

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

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

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

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

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

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

The first question is discussed in the section *Calling assembler routines from C*, page 167. The following two are covered in the section *Calling convention*, page 170.

For information about how data in memory is accessed, see *Memory access methods*, page 180.

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 181.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 167, and *Calling assembler routines from C++*, page 169, respectively.

## INLINE ASSEMBLER

Inline assembler can be used for inserting assembler instructions directly into a C or C++ function. Typically, this can be useful if you need to:



- Access hardware resources that are not accessible in C (in other words, when there is no definition for an SFR or there is no suitable intrinsic function available).
- Manually write a time-critical sequence of code that if written in C will not have the right timing.
- Manually write a speed-critical sequence of code that if written in C will be too slow.

An inline assembler statement is similar to a C function in that it can take input arguments (input operands), have return values (output operands), and read or write to C symbols (via the operands). An inline assembler statement can also declare *clobbered resources* (that is, values in registers and memory that have been overwritten).

### Limitations

Most things you can do in normal assembler language are also possible with inline assembler, with the following differences:

- Alignment cannot be controlled; this means, for example, that `DC32` directives might be misaligned.
- The only accepted register synonyms are `SP` (for `R13`), `LR` (for `R14`), and `PC` (for `R15`).
- In general, assembler directives will cause errors or have no meaning. However, data definition directives will work as expected.
- Resources used (registers, memory, etc) that are also used by the C compiler must be declared as operands or clobbered resources.
- If you do not want to risk that the inline assembler statement to be optimized away by the compiler, you must declare it `volatile`.
- Accessing a C symbol or using a constant expression requires the use of operands.
- Dependencies between the expressions for the operands might result in an error.

### Risks with inline assembler

Without operands and clobbered resources, inline assembler statements have no interface with the surrounding C source code. This makes the inline assembler code fragile, and might also become a maintenance problem if you update the compiler in the future. There are also several limitations to using inline assembler without operands and clobbered resources:

- The compiler's various optimizations will disregard any effects of the inline statements, which will not be optimized at all.
- The inline assembler statement will be `volatile` and *clobbered memory* is not implied. This means that the compiler will not remove the assembler statement. It will simply be inserted at the given location in the program flow. The consequences

or side-effects that the insertion might have on the surrounding code are not taken into consideration. If, for example, registers or memory locations are altered, they might have to be restored within the sequence of inline assembler instructions for the rest of the code to work properly.



The following example demonstrates the risks of using the `asm` keyword without operands and clobbers:

```
int Add(int term1, int term2)
{
    asm("add r0,r0,r1");
    return term1;
}
```

In this example:

- The function `Add` assumes that values are passed and returned in registers in a way that they might not always be, for example if the function is inlined.
- The `add` instruction updates the condition flags, which should be specified using the `cc` clobber operand. Otherwise, the compiler will assume that the condition flags are not modified.

Inline assembler without using operands or clobbered resources is therefore often best avoided.

## Reference information for inline assembler

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

### Syntax

The syntax of an inline assembler statement is (similar to the one used by GNU `gcc`):

```
asm [volatile]( string [assembler-interface])
```

*string* can contain one or more valid assembler instructions or data definition assembler directives, separated by `\n`.

For example:

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

Note that you can define and use local labels in inline assembler instructions.

*assembler-interface* is:

```

: comma-separated list of output operands      /* optional */
: comma-separated list of input operands       /* optional */
: comma-separated list of clobbered resources /* optional */

```

## Operands

An inline assembler statement can have one input and one output comma-separated list of operands. Each operand consists of an optional symbolic name in brackets, a quoted constraint, followed by a C expression in parentheses.

## Syntax of operands

```
[ [ symbolic-name ] ] " [modifiers] constraint" (expr)
```

For example:

```

int Add(int term1, int term2)
{
    int sum;

    asm("add %0,%1,%2"
        : "=r"(sum)
        : "r" (term1), "r" (term2)
        : "cc");

    return sum;
}

```

In this example, the assembler instruction uses one output operand, `sum`, two input operands, `term1` and `term2`, and clobbers the condition flags.

It is possible to omit any list by leaving it empty. For example:

```

int matrix[M][N];

void MatrixPreloadRow(int row)
{
    asm volatile ("pref %0 [0]" : : "r" (&matrix[row][0]));
}

```

## Operand constraints

Constraint	Description
<code>r</code>	Uses a general purpose register for the expression: R0-R12, R14
<code>Rp</code>	Uses an aligned pair of general purpose register for the expression: R1:R0-R11:R10
<code>i</code>	A symbolic or numerical constant. Only valid for input operands.

Table 21: Inline assembler operand constraints

## Constraint modifiers

Constraint modifiers can be used together with a constraint to modify its meaning. This table lists the supported constraint modifiers:

Modifier	Description
=	Write-only operand
+	Read-write operand
&	Early clobber output operand which is written to before the instruction has processed all the input operands.

Table 22: Supported constraint modifiers

## Referring to operands

Assembler instructions refer to operands by prefixing their order number with %. The first operand has order number 0 and is referred to by %0.

If the operand has a symbolic name, you can refer to it using the syntax %[*operand.name*]. Symbolic operand names are in a separate namespace from C/C++ code and can be the same as a C/C++ variable names. Each operand name must however be unique in each assembler statement. For example:

```
int Add(int term1, int term2)
{
    int sum;

    asm("add %[Rd], %[Rn], %[Rm] "
        : [Rd]"=r"(sum)
        : [Rn]"r"(term1), [Rm]"r"(term2)
        : "cc");

    return sum;
}
```

## Operand modifiers

An operand modifier is a single letter between the % and the operand number, which is used for transforming the operand.

In the example below, the modifiers L and H are used for accessing the least and most significant register, respectively, of a register pair:

```
unsigned long long Mul64x64(unsigned long long x, unsigned long
long y)
{
    unsigned long long res;
    asm("mulu.d %0,%L1,%L2\n"
        "mac    %M0,%L1,%M2\n"
        "mac    %M0,%M1,%L2\n"
        : "&Rp" (res)
        : "Rp" (x), "Rp" (y));
    return res;
}
```

Some operand modifiers can be combined, in which case each letter will transform the result from the previous modifier. This table describes the transformation performed by each valid modifier:

Modifier	Description
L	The least significant register of a register pair.
M	The most significant register of a register pair.
Q	The quotient part of a register pair as defined by <code>DIV[S U]</code> .
R	The result part of a register pair as defined by <code>DIV[S U]</code> .

Table 23: Operand modifiers and transformations

Input operands	<p>Input operands cannot have any modifiers, but they can have any valid C expression as long as the type of the expression fits the register.</p> <p>The C expression will be evaluated just before any of the assembler instructions in the inline assembler statement and assigned to the constraint, for example a register.</p>
Output operands	<p>Output operands must have <code>=</code> as a modifier and the C expression must be an l-value and specify a writable location. For example, <code>=r</code> for a write-only general purpose register. The constraint will be assigned to the evaluated C expression (as an l-value) immediately after the last assembler instruction in the inline assembler statement. Input operands are assumed to be consumed before output is produced and the compiler may use the same register for an input and output operand. To prohibit this, prefix the output constraint with <code>&amp;</code> to make it an early clobber resource, for example <code>=&amp;r</code>. This will ensure that the output operand will be allocated in a different register than the input operands.</p>
Input/output operands	<p>An operand that should be used both for input and output must be listed as an output operand and have the <code>+</code> modifier. The C expression must be an l-value and specify a writable location. The location will be read immediately before any assembler instructions and it will be written to right after the last assembler instruction.</p> <p>This is an example of using a read-write operand:</p> <pre>int Double(int value) {     asm("add %0,%0,%0" : "+r"(value) : : "cc");      return value; }</pre> <p>In the example above, the input value for <code>value</code> will be placed in a general purpose register. After the assembler statement, the result from the <code>ADD</code> instruction will be placed in the same register.</p>

**Clobbered resources**

An inline assembler statement can have a list of clobbered resources.

```
"resource1", "resource2", ...
```

Specify clobbered resources to inform the compiler about which resources the inline assembler statement destroys. Any value that resides in a clobbered resource and that is needed after the inline assembler statement will be reloaded.

Clobbered resources will not be used as input or output operands.

This is an example of how to use clobbered resources:

```
int Add(int term1, int term2)
{
    int sum;

    asm("add %0,%1,%2"
        : "=r" (sum)
        : "r" (term1), "r" (term2)
        : "cc");

    return sum;
}
```

In this example, the condition codes will be modified by the `ADDS` instruction. Therefore, `"cc"` must be listed in the clobber list.

This table lists valid clobbered resources:

<b>Clobber</b>	<b>Description</b>
R0-R12, R14	General purpose registers.
cc	The condition flags (N, Z, V, and C).
memory	To be used if the instructions modify any memory. This will avoid keeping memory values cached in registers across the inline assembler statement.

*Table 24: List of valid clobbers*

## AN EXAMPLE OF HOW TO USE CLOBBERED MEMORY

```
int StoreExclusive(unsigned long * location, unsigned long value)
{
    int failed;

    asm("stcond %1 [0],%2\n"
        "srne  %0"
        : "=&r" (failed)
        : "r" (location), "r" (value)
        : "memory");

    return failed;
}
```

---

## Calling assembler routines from C

An assembler routine that will be called from C must:

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

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

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

### CREATING SKELETON CODE

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

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the

variables required and perform simple accesses to them. In this example, the assembler routine takes an `int` and a `char`, and then returns an `int`:

```
extern int gInt;
extern char gChar;

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

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

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

## COMPILING THE SKELETON CODE



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



Use these options to compile the skeleton code:

```
iccavr32 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 `s82`. 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.s82`.



**Note:** The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file.



In the IDE, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**.



On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY **Call Stack** window work.

### The output file

The output file contains the following important information:

- The calling convention
- The return values
- The global variables
- The function parameters
- How to create space on the stack (auto variables)
- Call frame information (CFI).

The CFI directives describe the call frame information needed by the **Call Stack** window in the debugger. For more information, see *Call frame information*, page 181.

---

## Calling assembler routines from C++

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

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

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

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

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

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

```
class MyClass;

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

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

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

- Assembler list files resulting from compiling C++ files cannot, in general, be passed through the assembler.
- It is not possible to refer to or define C++ functions that do not have C linkage in assembler.

---

## Calling convention

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

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

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

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

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

The compiler follows the ABI defined by the procedure call standard for the AVR32 microprocessor. ABI stands for Application Binary Interface and it defines how functions and modules should interact.

## FUNCTION DECLARATIONS

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

```
int MyFunction(int first, char * second);
```

This means that the function takes two parameters: an integer and a pointer to a character. The function returns a value, an integer.

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

## USING C LINKAGE IN C++ SOURCE CODE

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

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

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

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

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

int F(int);

#ifdef __cplusplus
}
#endif
```

## PRESERVED VERSUS SCRATCH REGISTERS

The general AVR32 CPU registers are divided into three separate sets, which are described in this section.

### Scratch registers

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

Any of the registers R8 to R12 can be used as a scratch register by the function. This means that the original value does not have to be preserved. However, there is one exception: A return value pointer in R12 must be preserved.

### Preserved registers

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

The registers R0 through to R7 are preserved registers, as well as a return value pointer in R12.

### Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer register must at all times point to or below the last element on the stack. In the eventuality of an interrupt, everything below the point the stack pointer points to, will be destroyed.
- The return address register LR holds the return address at the entrance of the function.

## FUNCTION ENTRANCE

Parameters can be passed to a function using one of these basic methods:

- In registers
- On the stack

It is much more efficient to use registers than to take a detour via memory, so the calling convention is designed to use registers as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. The parameters are also passed on the stack in these cases:

- Aggregate types—such as structures, unions, arrays, and classes—unless their size and alignment have a matching scalar type
- Unnamed parameters to variable length (variadic) functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

### Note:

- Interrupt functions cannot take any parameters.
- For `__scall` declared functions, all parameters must be passed in registers. If any parameter does not fit in the available registers, an error message will be issued.

## Hidden parameters

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

- If the function returns a structure that does not fulfill the conditions specified in *Function entrance*, page 173, the memory location where to store the structure is passed in the register `R12` as a hidden parameter.
- If the function is a non-static Embedded C++ member function, then the `this` pointer is passed as a parameter. The reason for the requirement that the member function must be non-static is that static member methods do not have a `this` pointer.

**Note:** The register used for passing the `this` pointer is an internal property of the compiler and as such subject to change.

## Register parameters

The registers available for passing parameters are R12–R8, in that order:

Parameters	Passed in registers
8-bit values	R12, R11, R10, R9, R8
16-bit values	R12, R11, R10, R9, R8
24-bit values	R12, R11, R10, R9, R8
32-bit values	R10 : R11, R8 : R9

*Table 25: Registers used for passing parameters*

**Note:** When a parameter is passed in a register pair, the most significant word is placed in the register with the highest number, R11 or R9.

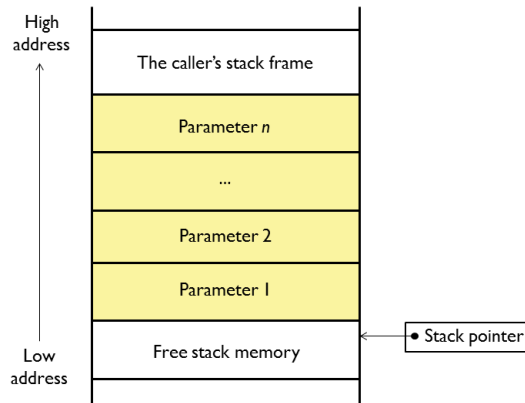
The procedure call standard defines the following process for assigning registers to parameters:

The assignment of registers to parameters is a straightforward process. Traversing the parameters from left to right (first to last), the parameter is assigned to the first available register or register pair. Should there be no suitable register available, the parameter is passed on the stack. All parameters are considered until no more parameters are left.

## Stack parameters and layout

Stack parameters are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer (toward low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack that is divisible by four, etc.

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 `R12` and `R10:R11`:

Return values	Passed in registers
8-bit values (if word alignment $\geq 1$ )	<code>R12</code>
16-bit values (if word alignment $\geq 2$ )	<code>R12</code>
32-bit values (if word alignment $\geq 4$ or unaligned word access is enabled)	<code>R12</code>
64-bit values (if word alignment $\geq 4$ )	<code>R10:R11</code>

Table 26: Registers used for returning values

When returning an integer or pointer in `R12`, the status flags are also set according to a `cp.w R12, 0` instruction.

## Stack layout at function exit

It is the responsibility of the caller to clean the stack after the called function has returned.

## 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 in the return address register LR.

Typically, a function returns by using the `ret` or `popm` instruction, for example:

```
ret    r12
```

If a function is to call another function, the original return address must be stored somewhere. This is normally done on the stack, for example:

```
name    call
section CODE:CODE
extern  func

st.w    --sp,lr
rcall   func
ld.w    lr,sp++

; Do something here.

ret
end
```

The return address is restored directly from the stack with the `popm` or `ldm` instructions.

## CALLS IN SUPERVISOR MODE

When a call is made to a function declared `__scall`, the calling convention described above cannot be followed.

Parameters cannot be passed on the stack, because the stack used before the call (the application stack) is not the same as the stack used inside the `__scall` function (the system stack).

This means that the following restrictions apply when calling an `__scall` function:

- Parameters will be assigned to registers, just as for a call to a normal function, but only the registers R12–R8 are available
- All parameters must be passed in registers. If any parameter does not fit in the available registers, an error message will be issued.

## ALTERNATIVE CALLING CONVENTION FOR FLASHVAULT IMPLEMENTATION FUNCTIONS

A function declared with the extended keyword `__flashvault_impl` adjusts the calling convention of the function by inserting 4 extra bytes on the stack, placed after (below) all stack parameters.



These 4 extra bytes are used by the secure state API handler to save the actual return address before calling the API implementation function. The compiler automatically adjusts the stack when a `__flashvault_impl` declared function is called directly, that is from within the secure state code.

For more information, see *Implementing middleware using FlashVault™*, page 84.

## EXAMPLES

The following section shows a series of declaration examples and the corresponding calling conventions. The complexity of the examples increases toward the end.

### Example 1

Assume this function declaration:

```
int add1(int);
```

This function takes one parameter in the register R12, and the return value is passed back to its caller in the register R12.

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

```
name      return
section  CODE:CODE
sub      r12, -1
ret      r12
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 integer parameter `y` is passed in the register R12 and the structure parameter `x` is passed on the stack. The return value is passed back to its caller in the register R12.

### Example 3

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

```

struct MyStruct
{
    int mA[20];
};

struct MyStruct MyFunction(int x);

```

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in `R12`. The caller assumes that these registers remain untouched. The parameter `x` is passed in `R11`.

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

### FUNCTION DIRECTIVES

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

For more information about the function directives, see the *IAR Assembler User Guide for AVR32*.

---

## Assembler instructions used for calling functions

This section presents the assembler instructions that can be used for calling and returning from functions on the AVR32 microprocessor.

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

The normal function calling instructions are the relative call, the memory call, and the pseudo-memory call instructions:

```

        rcall func
        mcall ??func
        _pcall func,??func

        align 2
??func    dc32 func

```

The location that the called function should return to (that is, the location immediately after this instruction) is stored in the link register, `lr`.

The pseudo-instruction `_pcall` will either assemble as an `rcall` instruction, if the destination label is within  $\pm 1$ Mbyte of the call site, or as an `mcall` instruction using a constant table entry to hold a full 32-bit address. Note that the table entry will always be generated even if an `rcall` instruction is generated.

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

## CALLING FUNCTIONS IN THE SMALL AND MEDIUM CODE MODELS

A direct call using these code models is simply:

```
rcall function
```

The relative call instruction, `rcall`, can only reach  $\pm 1$ Mbyte, which effectively limits the application size to 1 Mbyte. In the small code model, the `__data32` keyword can be used for forcing a long call (the `_pcall` pseudo-instruction) to be used.

The main difference between the small and medium code models are that the small code model uses the `mov`, `sub`, and `cp` instructions to handle function addresses, while the medium code model uses constant table entries or a combination of `mov` and `orh` instructions:

```

mov     r12,lwrd(x)
orh     r12,hwrd(x)

```

## CALLING FUNCTIONS IN THE LARGE CODE MODEL

A direct call using this code model is:

```

        _pcall function,??func
        ...
        align 2
??func    dc32 function

```

The `_pcall` pseudo-instruction will generate either an `rcall` or an `mcall` instruction, depending on the distance to the called function.

When a function call is made via a function pointer—an indirect function call—this code will be generated:

```

name      callFuncPtr
segment   CODE:CODE
extern    funcPtr

mov       r12,funcPtr      ; Location of function pointer
ld.w     r11,r12[0]       ; Load function address
icall    r11               ; 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.

---

## 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 AVR32 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 MAIN MEMORY ACCESS METHOD (DATA21, DATA32)

Most instructions can address the entire main memory, and can be used irrespective of whether the variable has been declared with the `__data21` keyword or the `__data32` keyword (in analogy with using the Small or the Large data model). The only difference is that some instructions can handle a short direct address directly, like `mov Rd, address, cp Rd, address, and sub Rd, address.`

An indirect access is always equivalent irrespective of which of the two main memory keywords that has been used to declare a variable.

Both a pointer and the index type of an array have a size of 32 bits.

## READ-MODIFY-WRITE ACCESS METHOD (DATA17)

If the processor supports the RMW instruction set extension and you have enabled the use of these instructions (see `--avr32_rmw_instructions`, page 190), the variables declared in data17 memory can use these instructions for efficient bit manipulation. There are three instructions in the RMW instruction set extension: `memc` which clears a bit, `mems` which sets a bit, and `memt` which toggles a bit.

There is no reason to use the `__data17` attribute when the RMW instruction set extension is not available.

## THE SYSTEM AND DEBUG REGISTER ACCESS METHOD

To access variables declared with the `__sysreg` keyword, the special instructions `mfsr` and `mtsr` must be used. For variables declared `__dbgreg`, the special instructions `mtdr` and `mfdcr` must be used. Only direct accesses are possible, as you cannot take the address of a system register variable.

### Examples

These examples access data17 memory in different ways:

```

mems    x,31           ; Direct access of x.

mov     r9,y           ; Access the global variable y.
ld.b    r8,r9[0]

mov     r12,z          ; Access an array
ld.sh   r10,r12[r11<<1]

```

---

## 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 AVR32*.

## 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 frame on the stack. It is associated with the contents of the stack pointer <code>SP</code> .
RAR	The return address register which holds the return address for interrupt and exception handlers.
RAR_DBG	The return address register which holds the return address for debug exception functions.
?RET	The return address. The return address is initially found in the register <code>LR</code> , but can be moved on the stack later on.
RSR	The return status register which holds a copy of the status register that will be restored at exit from interrupt and exception handlers.

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

Resource	Description
RSR_DBG	The return status register which holds a copy of the status register that will be restored at exit from a debug exception handler.
R0–R12	The ordinary data registers.
SP	The stack pointer.
SR	The status register.
LR	The link register.

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

## 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 `-lA`.



In the IDE, choose **Project>Options>C/C++ Compiler>List** and make sure the suboption **Include call frame information** is selected.

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

```

NAME Cfi

RTMODEL "__SystemLibrary", "DLib"
RTMODEL "__code_model", "medium"
RTMODEL "__core", "avr32a"
RTMODEL "__data_model", "small"
RTMODEL "__enum_size", "fixed"
RTMODEL "__rt_version", "2"
RTMODEL "__unaligned_word_access", "disabled"

RSEG CSTACK:DATA:REORDER:NOROOT(2)

PUBLIC cfiExample

FUNCTION cfiExample,021203H
ARGFRAME CSTACK, 0, STACK
LOCFRAME CSTACK, 8, STACK

CFI Names cfiNames0
CFI StackFrame CFA SP DATA
CFI VirtualResource ?RET:32
CFI Resource R0:32, R1:32, R2:32, R3:32, R4:32, R5:32,
R6:32, R7:32
CFI Resource RSR:32, RAR:32, RSR_DBG:32, RAR_DBG:32
CFI EndNames cfiNames0

CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 2
CFI DataAlign 4
CFI ReturnAddress ?RET CODE
CFI CFA SP+0
CFI ?RET LR
CFI R0 SameValue
CFI R1 SameValue
CFI R2 SameValue
CFI R3 SameValue
CFI R4 SameValue
CFI R5 SameValue
CFI R6 SameValue
CFI R7 SameValue

```



```

CFI R8 Undefined
CFI R9 Undefined
CFI R10 Undefined
CFI R11 Undefined
CFI R12 Undefined
CFI LR Undefined
CFI SR SameValue
CFI RSR SameValue
CFI RAR SameValue
CFI RSR_DBG SameValue
CFI RAR_DBG SameValue
CFI EndCommon cfiCommon0

EXTERN F
FUNCTION F,0202H

RSEG CODE32:CODE:REORDER:NOROOT(2)
CFI Block cfiBlock0 Using cfiCommon0
CFI Function cfiExample
CODE
cfiExample:
    FUNCALL cfiExample, F
    LOCFRAME CSTACK, 8, STACK
    STM      --SP,R7,LR
    CFI R7 Frame(CFA, -4)
    CFI ?RET Frame(CFA, -8)
    CFI CFA SP+8
    MOV      R7,R12
    MOV      R12,R7
    RCALL    F:E
    ADD      R7,R12
    MOV      R12,R7
    LDM      SP++,R7,PC
    CFI EndBlock cfiBlock0

END

```

**Note:** The header file `cfi.m82` contains the macros `CFI_NAMES` and `CFI_COMMON_DEFAULT`, 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 AVR32 supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

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

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

- The `inline` keyword advises the compiler that the function defined immediately after the keyword should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a `for` loop
- The `bool` data type
- The `long long` data type
- The `complex` floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator `_Pragma()`
- Variadic macros, which are the preprocessor macro equivalents of `printf` style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option `--vla`

- Inline assembler using the `asm` or the `__asm` keyword, see .

**Note:** Even though it is a C99 feature, the IAR C/C++ Compiler for AVR32 does not support UCNs (universal character names).

---

## 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 189. For more information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

The `#pragma` directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For information about available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

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

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. For more information about using intrinsic functions, see *Mixing C and assembler*, page 159. For information about available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides the C and C++ library definitions that apply to embedded systems. For more information, see *IAR DLIB Library*, page 357.

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

## ENABLING LANGUAGE EXTENSIONS

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

Command line	IDE*	Description
<code>--strict</code>	<b>Strict</b>	All IAR C language extensions are disabled; errors are issued for anything that is not part of Standard C.
None	<b>Standard</b>	All <i>extensions to Standard C</i> are enabled, but no <i>extensions for embedded systems programming</i> . For information about extensions, see <i>IAR C language extensions</i> , page 189.
<code>-e</code>	<b>Standard with IAR extensions</b>	All <i>IAR C language extensions</i> are enabled.

Table 28: Language extensions

\* In the IDE, choose **Project>Options>C/C++ Compiler>Language>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

## IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific microprocessor 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 192.

## EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

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

- Memory attributes, type attributes, and object attributes
 

For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.
- Placement at an absolute address or in a named segment
 

The @ operator or the directive `#pragma location` can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these features, see *Controlling data and function placement in memory*, page 214, and *location*, page 318.
- Alignment control
 

Each data type has its own alignment; for more information, see *Alignment*, page 275. 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.
- Anonymous structs and unions
 

C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 213.
- 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 277.
- `static_assert()`

The construction `static_assert(const-expression, "message");` can be used in C/C++. The construction will be evaluated at compile time and if `const-expression` is false, a message will be issued including the `message` string.

- Parameters in variadic macros

Variadic macros are the preprocessor macro equivalents of `printf` style functions. The preprocessor accepts variadic macros with no arguments, which means if no parameter matches the `...` parameter, the comma is then deleted in the `#, ##__VA_ARGS__` macro definition. According to Standard C, the `...` parameter must be matched with at least one argument.

### Dedicated segment operators

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

<code>__segment_begin</code>	Returns the address of the first byte of the named segment.
<code>__segment_end</code>	Returns the address of the first byte <i>after</i> the named segment.
<code>__segment_size</code>	Returns the size of the named segment in bytes.

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

These operators behave syntactically as if declared like:

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

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

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

#### Example

In this example, the type of the `__segment_begin` operator is `void __data17 *`.

```
#pragma segment="MYSEGMENT" __data17
...
segment_start_address = __segment_begin("MYSEGMENT");
```

See also *segment*, page 324, and *location*, page 318.

## 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 282.
- Taking the address of a register variable
 

In Standard C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.
- `long float` means `double`

The type `long float` is accepted as a synonym for `double`.
- Repeated `typedef` declarations
 

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.
- Mixing pointer types
 

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

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.



- Non-top level `const`  
Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.
- Non-lvalue arrays  
A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.
- Comments at the end of preprocessor directives  
This extension, which makes it legal to place text after preprocessor directives, is enabled unless the strict Standard C mode is used. The purpose of this language extension is to support compilation of legacy code; we do not recommend that you write new code in this fashion.
- An extra comma at the end of `enum` lists  
Placing an extra comma is allowed at the end of an `enum` list. In strict Standard C mode, a warning is issued.
- A label preceding a `}`  
In Standard C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler allows this, but issues a warning.  
Note that this also applies to the labels of `switch` statements.
- Empty declarations  
An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).
- Single-value initialization  
Standard C requires that all initializer expressions of static arrays, structs, and unions are enclosed in braces.  
Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:  

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

- Declarations in other scopes

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

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

- Expanding function names into strings with the function as context

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

```
"void func(char)"
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 254.

- Static functions in function and block scopes

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

- Numbers scanned according to the syntax for numbers

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

# Using C++

- Overview—EC++ and EEC++
- Enabling support for C++
- EC++ feature descriptions
- EEC++ feature description
- C++ language extensions

---

## Overview—EC++ and EEC++

IAR Systems supports the C++ language. You can choose between the industry-standard Embedded C++ and Extended Embedded C++. *Using C++* describes what you need to consider when using the C++ language.

Embedded C++ is a proper subset of the C++ programming language which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language. EC++ offers the same object-oriented benefits as C++, but without some features that can increase code size and execution time in ways that are hard to predict.

### EMBEDDED C++

These C++ features are supported:

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

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are features added very late before Standard C++ was defined. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

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

- Templates
- Multiple and virtual inheritance
- Exception handling
- Runtime type information
- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

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

## EXTENDED EMBEDDED C++

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

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

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

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++

language, which means no exceptions and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

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

---

## Enabling support for C++



In the compiler, the default language is C.

To compile files written in Embedded C++, use the `--ec++` compiler option. See `--ec++`, page 255.

To take advantage of *Extended* Embedded C++ features in your source code, use the `--eec++` compiler option. See `--eec++`, page 255.



To enable EC++ or EEC++ in the IDE, choose **Project>Options>C/C++ Compiler>Language** and select the appropriate standard.

---

## EC++ feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for AVR32, 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 __sysreg memory at address 60
    static __sysreg __no_init int mI @ 60;

    // Locate a static function in __code21 memory
    static __code21 void F();

    // Locate a function in __code21 memory
    __code21 void G();

    // Locate a virtual function in __code21 memory
    virtual __code21 void H();

    // Locate a virtual function into SPECIAL
    virtual void M() const volatile @ "SPECIAL";
};

```

### 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
}

```

## 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 131.

Function local static class objects are constructed the first time execution passes through their declaration, and are destroyed when returning from `main` or when calling `exit`.

## USING NEW HANDLERS

To handle memory exhaustion, you can use the `set_new_handler` function.

### New handlers in Embedded C++

If you do not call `set_new_handler`, or call it with a NULL new handler, and `operator new` fails to allocate enough memory, it will call `abort`. The `nothrow` variant of the new operator will instead return NULL.

If you call `set_new_handler` with a non-NULL new handler, the provided new handler will be called by `operator new` if `operator new` fails to allocate memory. The new handler must then make more memory available and return, or abort execution in some manner. The `nothrow` variant of `operator new` will never return NULL in the presence of a new handler.

## TEMPLATES

Extended EC++ supports templates according to the C++ standard, but not the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

## DEBUG SUPPORT IN C-SPY

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

For more information about this, see the *C-SPY® Debugging Guide for AVR32*.

---

## EEC++ feature description

This section describes features that distinguish Extended EC++ from EC++.

### TEMPLATES

The compiler supports templates with the syntax and semantics as defined by Standard C++. However, note that the STL (standard template library) delivered with the product is tailored for Extended EC++, see *Extended Embedded C++*, page 196.

### VARIANTS OF CAST OPERATORS

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

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

### MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

### NAMESPACE

The namespace feature is only supported in *Extended EC++*. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

### THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.



## C++ language extensions

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

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

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

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

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

According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:

```
struct A
{
    int A::F(); // Possible when using IAR language extensions
    int G();   // According to the standard
};
```

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

```
extern "C" void F(); // Function with C linkage
void (*PF)()        // PF points to a function with C++ linkage
                   = &F; // Implicit conversion of function pointer.
```

According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```
bool X;

char *P1 = X ? "abc" : "def";           //Possible when using IAR
                                         //language extensions
char const *P2 = X ? "abc" : "def"; //According to the standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.
- An anonymous union can be introduced into a containing class by a `typedef` name. It is not necessary to first declare the union. For example:

```
typedef union
{
    int i,j;
} U; // U identifies a reusable anonymous union.

class A
{
public:
    U; // OK -- references to A::i and A::j are allowed.
};
```

In addition, this extension also permits *anonymous classes* and *anonymous structs*, as long as they have no C++ features (for example, no static data members or member functions, and no non-public members) and have no nested types other than other anonymous classes, structs, or unions. For example:

```
struct A
{
    struct
    {
        int i,j;
    }; // OK -- references to A::i and A::j are allowed.
};
```

- The friend class syntax allows nonclass types as well as class types expressed through a `typedef` without an elaborated type name. For example:

```
typedef struct S ST;

class C
{
public:
    friend S; // Okay (requires S to be in scope)
    friend ST; // Okay (same as "friend S;")
    // friend S const; // Error, cv-qualifiers cannot
                       // appear directly
};
```

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



# Application-related considerations

- Stack considerations
- Heap considerations
- Interaction between the tools and your application
- Checksum calculation

---

## 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 224.

---

## Heap considerations

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

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

- Linker segments used for the heap

- Allocating the heap size, see *Setting up heap memory*, page 109.

## HEAP SEGMENTS IN DLIB

The memory allocated to the heap is placed in the segment `HEAP`, which is only included in the application if dynamic memory allocation is actually used.

## HEAP SIZE 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 AVR32 microprocessor. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

---

## Interaction between the tools and your application

The linking process and the application can interact symbolically in four ways:

- Creating a symbol by using the linker command line option `-D`. The linker will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.
- Using the compiler operators `__segment_begin`, `__segment_end`, or `__segment_size`, or the assembler operators `SFB`, `SFE`, or `SIZEOF` on a named segment. These operators provide access to the start address, end address, and size of a contiguous sequence of segments with the same name
- The command line option `-s` informs the linker about the start label of the application. It is used by the linker as a root symbol and to inform the debugger where to start execution.

The following lines illustrate how to use `-D` to create a symbol. If you need to use this mechanism, add these options to your command line like this:

```
-Dmy_symbol=A
-DMY_HEAP_SIZE=400
```

The linker configuration file can look like this:

```
-Z (DATA) MyHeap+MY_HEAP_SIZE=20000-2FFFF
```

Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by an XLINK option to dynamically allocate
an array of elements with specified size. The value takes the
form of a label.
*/
extern int NrOfElements;

typedef char Elements;
Elements *GetElementArray()
{
    return malloc(sizeof(Elements) * (long) &NrOfElements);
}

/* Use a symbol defined by an XLINK option, a symbol that in the
* configuration file was made available to the application.
*/
extern char HeapSize;

/* Declare the section that contains the heap. */
#pragma segment = "MYHEAP"

char *MyHeap()
{
    /* First get start of statically allocated section, */
    char *p = __segment_begin("MYHEAP");

    /* ...then we zero it, using the imported size. */
    for (int i = 0; i < (int) &HeapSize; ++i)
    {
        p[i] = 0;
    }
    return p;
}
```

---

## Checksum calculation

To use checksumming to verify the integrity of your application, you must:

- Choose a checksum algorithm by setting the command line option `-J`, and include source code for the algorithm in your application
- Decide which memory ranges to verify and set up the linker by using the command line option `-J`, and the source code for it in your application source code.

- Make sure your application refers to the checksum symbol (see `-J` in the *IAR Linker and Library Tools Reference Guide*) to ensure that is included.



In the IDE, choose **Project>Options>Linker>Checksum**.

## CALCULATING A CHECKSUM

In this example, a checksum is calculated for ROM memory at `0x8002` up to `0x8FFF` and the 2-byte calculated checksum is placed at `0x8000`.

## ADDING A CHECKSUM FUNCTION TO YOUR SOURCE CODE

To check the value of the generated checksum, it must be compared with a checksum that your application calculated. This means that you must add a function for checksum calculation (that uses the same algorithm as the `-J` option) to your application source code. Your application must also include a call to this function.

### A function for checksum calculation

This function—a slow variant but with small memory footprint—uses the `crc16` algorithm:

```
unsigned short SlowCrc16(unsigned short 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 the checksum algorithms in the `avr32\src\linker` directory of your product installation.



## Example of checksum calculation

This code gives an example of how the checksum can be calculated:

```

/* The checksum calculated
 * (note that it is located on address 0x8000)
 */
extern unsigned short const __checksum;

void TestChecksum()
{
    unsigned short calc = 0;
    unsigned char zeros[2] = {0, 0};

    /* Run the checksum algorithm */
    calc = SlowCrc16(0,
                    (unsigned char *) checksumStart,
                    (checksumEnd - checksumStart+1));

    /* Rotate out the answer */
    calc = SlowCrc16(calc, zeros, 2);

    /* Test the checksum */
    if (calc != __checksum)
    {
        abort(); /* Failure */
    }
}

```

## THINGS TO REMEMBER

When calculating a checksum, you must remember that:

- Typically, the checksum must be calculated from the lowest to the highest address for every memory range
- Each memory range must be verified in the same order as defined (ABC is not the same as ACB)
- It is OK to have several ranges for one checksum
- If several checksums are used, you should place them in sections with unique names and use unique symbol names
- If the a slow function variant is used, you must make a final call to the checksum calculation with as many bytes (with the value 0x00) as there are bytes in the checksum.
- Never calculate a checksum on a location that contains a checksum.



# 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 32-bit data types (`signed int` or `unsigned int` or `long`), unless memory is restricted.
- Try to avoid 64-bit data types, such as `double` and `long long`.
- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array, that is `int`.
- 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 tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

### FLOATING-POINT TYPES

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

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

In the compiler, the floating-point type `float` always uses the 32-bit format, and the type `double` always uses the 64-bit format.

By default, a *floating-point constant* in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, the `double` constant `1.0` is added and the result is converted back to a `float`:

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

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

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

For more information about floating-point types, see *Basic data types—floating-point types*, page 279.

## ALIGNMENT OF ELEMENTS IN A STRUCTURE

The AVR32 microprocessor requires that when accessing data in memory, the data must be aligned. Each element in a structure must be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment correct.

There are situations when this can be a problem:

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

For information about alignment requirements, see *Alignment*, page 275.

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 321.

## 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 AVR32 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 254, for additional information.

### Example

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

```
struct S
{
    char mTag;
    union
    {
        long mL;
        float mF;
    };
} St;

void F(void)
{
    St.mL = 5;
}
```

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

```
__no_init volatile
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char way: 1;
        unsigned char out: 1;
    };
} @ 0xFFFF00100;

/* The variables are used here. */
void Test(void)
{
    IOPORT = 0;
    way = 1;
    out = 1;
}
```

This declares an I/O register byte IOPORT at address 0xFFFF00100. 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 71.

- 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 66.
- Memory attributes
 

Using IAR-specific keywords or pragma directives, you can override the default placement of functions and data objects. For more information, see *Using function memory attributes*, page 72 and *Using data memory attributes*, page 63.
- 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 215.
- The @ operator and the #pragma location directive for segment placement.
 

Using the @ operator or the #pragma location directive, you can place individual functions, variables, and constants in named segments. The placement of these segments can then be controlled by linker directives. For more information, see *Data and function placement in segments*, page 217

## 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.

- \_\_no\_init
- \_\_no\_init and const (without initializers)
- const (with initializers)

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

**Note:** All declarations of variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

## Examples

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

```
__no_init volatile char alpha @ 0xFFFF2000; /* OK */
```

The next example contains two `const` declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0xFFFF2004
__no_init const int beta; /* OK */

const int gamma @ 0xFFFF2008 = 3; /* OK */
```

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

This shows incorrect usage:

```
int delta @ 0xFFFF200C; /* Error, neither */
/* "__no_init" nor "const". */
__no_init int epsilon @ 0xFFFF200D; /* Error, misaligned. */
```

## C++ considerations

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

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

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

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

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

**Note:** C++ static member variables can be placed at an absolute address just like any other static variable.



## DATA AND FUNCTION PLACEMENT IN SEGMENTS

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

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

C++ static member variables can be placed in named segments just like any other static variable.

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker configuration file using the `-Z` or the `-P` segment control directives.

**Note:** Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

The location of the segments can be controlled from the linker configuration file.

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

### Examples of placing variables in named segments

In the following examples, a data object is placed in a user-defined segment. If no memory attribute is specified, the variable will, like any other variable, be treated as if it is located in the default memory. Note that you must as always ensure that the segment is placed in the appropriate memory area when linking.

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

#pragma location="MY_CONSTANTS"
const int beta = 42; /* OK */

const int gamma @ "MY_CONSTANTS" = 17; /* OK */
int theta @ "MY_ZEROS"; /* OK */
int phi @ "MY_INITED" = 4711; /* OK */
```

The compiler will warn that segments that contain zero-initialized and initialized data must be handled manually. To do this, you must use the linker option `-Q` to separate the initializers into one separate segment and the symbols to be initialized to a different segment. You must then write source code that copies the initializer segment to the initialized segment, and zero-initialized symbols must be cleared before they are used.

As usual, you can use memory attributes to select a memory for the variable. Note that you must as always ensure that the segment is placed in the appropriate memory area when linking.

```
__data17 __no_init int alpha @ "MY_DATA17_NOINIT"; /* Placed in
                                                    data17*/
```

This example shows incorrect usage:

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

### Examples of placing functions in named segments

```
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:

```
__code21 void f(void) @ "MY_CODE21_FUNCTIONS";
```

---

## Controlling compiler optimizations

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

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

### SCOPE FOR PERFORMED OPTIMIZATIONS

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

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. See *optimize*, page 320, 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 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 260.

**Note:** Only one object file is generated, and thus all symbols will be part of that object file.

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the optimizations to functions and variables that are actually used. For more information, see *--discard\_unused\_publics*, page 253.

## 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 Conditional returns
Medium	Same as above, and: Live-dead analysis and optimization Code hoisting Register content analysis and optimization Common subexpression elimination Static clustering
High (Balanced)	Same as above, and: Peephole optimization Cross jumping Instruction scheduling (when optimizing for speed) Loop unrolling Function inlining Code motion Type-based alias analysis

Table 29: Compiler optimization levels

**Note:** Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 221.

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

## SPEED VERSUS SIZE

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

If you use the optimization level High speed, the `--no_size_constraints` compiler option relaxes the normal restrictions for code size expansion and enables more aggressive optimizations.

## FINE-TUNING ENABLED TRANSFORMATIONS

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

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Static clustering
- 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 261.

### Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most efficient for smaller loops, where the loop overhead can be a substantial part of the total loop body.

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

The compiler heuristically decides which loops to unroll. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

To disable loop unrolling, use the command line option `--no_unroll`, see `--no_unroll`, page 264.

### 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 87.

### 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 261.

### 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 264.

### Example

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

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

### 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 261.

### 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.

For more information about the command line option, see `--no_scheduling`, page 262.

---

## Facilitating good code generation

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

- Using efficient addressing modes
- Helping the compiler optimize

- Generating more useful error message.

## WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables (non-static). Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 222. 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 219.
- Avoid using inline assembler without operands and clobbered resources. Instead, use SFRs or intrinsic functions if available. Otherwise, use inline assembler *with* operands and clobbered resources 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 will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

## 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 32-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 `0x00000080`, and `~0x00000080` becomes `0xFFFFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 8 bits set.

## PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code.

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 298.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 284.

## ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several AVR32 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, for example:

```
__no_init volatile union
{
    unsigned short mwctl2;
    struct
    {
        unsigned short edr: 1;
        unsigned short edw: 1;
        unsigned short lee: 2;
        unsigned short lemd: 2;
        unsigned short lepl: 2;
    } mwctl2bit;
} @ 0xFFF00100;

/* 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 */
    mwctl2 = 0x1234;

    /* Bitfield accesses */
    mwctl2bit.edw = 1;
    mwctl2bit.lepl = 3;
}
```

You can also use the header files as templates when you create new header files for other AVR32 devices. For information about the @ operator, see *Placing located data*, page 108.

## PASSING VALUES BETWEEN C AND ASSEMBLER OBJECTS

The following example shows how you in your C source code can use inline assembler to set and get values from a special purpose register:

```
unsigned long get_ACBA(void)
{
    unsigned long value;
    asm volatile("mfsr %0, 8 /* ACBA */ : "=r"(value));
    return value;
}

void set_ACBA(unsigned long value)
{
    asm volatile("mtsrl 8 /* ACBA */, %0" :: "r"(value));
}
```

The general purpose register is used for getting and setting the value of the special purpose register ACBA. The same method can be used also for accessing other special purpose registers and specific instructions.

To read more about inline assembler, see *Inline assembler*, page 160.

## NON-INITIALIZED VARIABLES

Normally, the runtime environment will initialize all global and static variables when the application is started.

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in a separate segment, according to the specified memory keyword. See the chapter *Linking overview* for more information.

For `__no_init`, the `const` keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

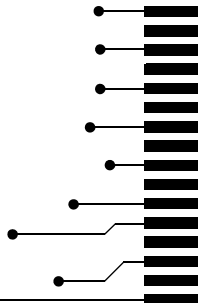
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For more information, see *\_\_no\_init*, page 300. Note that to use this keyword, language extensions must be enabled; see *-e*, page 254. For more information, see also *object\_attribute*, page 319.

# Part 2. Reference information

This part of the *IAR C/C++ Compiler User Guide for AVR32* contains these chapters:

- External interface details
- Compiler options
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- Segment reference
- The stack usage control file
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89.





# External interface details

- Invocation syntax
- Include file search procedure
- Compiler output
- Diagnostics

---

## Invocation syntax

You can use the compiler either from the IDE or from the command line. See the *IDE Project Management and Building Guide for AVR32* for information about using the compiler from the IDE.

### COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccavr32 [options] [sourcefile] [options]
```

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

```
iccavr32 prog.c --debug
```

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

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

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

## PASSING OPTIONS

There are three different ways of passing options to the compiler:

- Directly from the command line
  - Specify the options on the command line after the `iccavr32` command, either before or after the source filename; see *Invocation syntax*, page 231.
- Via environment variables
  - The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 232.
- Via a text file, using the `-f` option; see *-f*, page 256.

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

## ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
<code>C_INCLUDE</code>	Specifies directories to search for include files; for example: <code>C_INCLUDE=c:\program files\iar systems\embedded workbench .n\avr32\inc;c:\headers</code>
<code>QCCAVR32</code>	Specifies command line options; for example: <code>QCCAVR32=-lA asm.lst</code>

Table 30: Compiler environment variables

## Include file search procedure

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

- If the name of the `#include` file is an absolute path specified in angle brackets or double quotes, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:
 

```
#include <stdio.h>
```

 it searches these directories for the file to include:
  - 1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 258.
  - 2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 232.
  - 3 The automatically set up library system include directories. See *--dlib\_config*, page 253.



- 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:

```
iccavr32 ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

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

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

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

For information about the syntax for including header files, see *Overview of the preprocessor*, page 347.

---

## Compiler output

The compiler can produce the following output:

- A linkable object file

The object files produced by the compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `r82`.

- **Optional list files**  
Various kinds of list files can be specified using the compiler option `-l`, see `-l`, page 258. By default, these files will have the filename extension `lst`.
- **Optional preprocessor output files**  
A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `i`.
- **Diagnostic messages**  
Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 235.
- **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 234.
- **Size information**  
Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as *shared*.  
  
Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

## ERROR RETURN CODES

The compiler returns status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

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

Table 31: Error return codes

## Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

### MESSAGE FORMAT

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the compiler is produced in the form:

```
filename, linenumber level[tag]: message
```

with these elements:

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

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

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

### SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

#### Remark

A diagnostic message that is produced when the compiler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 269.

#### Warning

A diagnostic message that is produced when the compiler finds a potential programming error or omission which is of concern, but which does not prevent completion of the compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see `--no_warnings`, page 265.

## Error

A diagnostic message that is produced when the compiler finds a construct which clearly violates the C or C++ language rules, such that code cannot be produced. An error will produce a non-zero exit code.

## Fatal error

A diagnostic message that is produced when the compiler finds a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the message is issued, compilation terminates. A fatal error will produce a non-zero exit code.

## SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all diagnostics messages, except for fatal errors and some of the regular errors.

See the chapter *Compiler options*, for information about the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for information about the pragma directives that are available for setting severity levels.

## INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler. It is produced using this form:

Internal error: *message*

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Include enough information to reproduce the problem, typically:

- The product name
- The version number of the compiler, which can be seen in the header of the list files generated by the compiler
- Your license number
- The exact internal error message text
- The source file of the application that generated the internal error
- A list of the options that were used when the internal error occurred.

# Compiler options

- Options syntax
- Summary of compiler options
- Descriptions of compiler options

---

## Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IDE.



See the online help system for information about the compiler options available in the IDE and how to set them.

### TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it can have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 232.

### 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\`:

```
iccavr32 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:

```
iccavr32 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:
- ```
iccavr32 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:

```
iccavr32 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`:

```
iccavr32 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>--avr32_dsp_instructions</code>  | Enables <code>dsp</code> instructions  |
| <code>--avr32_flashvault</code>        | Enables secure mode instructions       |
| <code>--avr32_fpu_instructions</code>  | Enables <code>fpu</code> instructions  |
| <code>--avr32_rmw_instructions</code>  | Enables <code>rmw</code> instructions  |
| <code>--avr32_simd_instructions</code> | Enables <code>simd</code> instructions |

Table 32: Compiler options summary

| Command line option                                 | Description                                                                                                 |
|-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>--c89</code>                                  | Specifies the C89 dialect                                                                                   |
| <code>--char_is_signed</code>                       | Treats <code>char</code> as signed                                                                          |
| <code>--char_is_unsigned</code>                     | Treats <code>char</code> as unsigned                                                                        |
| <code>--code_model</code>                           | Specifies the code model                                                                                    |
| <code>--core</code>                                 | Specifies a CPU core                                                                                        |
| <code>--core_revision</code>                        | Specifies the revision of the CPU core                                                                      |
| <code>--cpu</code>                                  | Specifies a specific device                                                                                 |
| <code>--cpu_info</code>                             | Reads device-specific configuration information from a file                                                 |
| <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>--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_inline_asm_label_replacement</code> | Disables label replacement in inline assembler statements                                                   |
| <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>-e</code>                                     | Enables language extensions                                                                                 |
| <code>--ec++</code>                                 | Specifies Embedded C++                                                                                      |
| <code>--eec++</code>                                | Specifies Extended Embedded C++                                                                             |
| <code>--enable_multibytes</code>                    | Enables support for multibyte characters in source files                                                    |
| <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                                                                                    |

Table 32: Compiler options summary (Continued)



| Command line option                           | Description                                                                                                                      |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>--fp_implementation</code>              | Selects the floating-point implementation to use                                                                                 |
| <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>--library_module</code>                 | Creates a library module                                                                                                         |
| <code>--macro_positions_in_diagnostics</code> | Obtains positions inside macros in diagnostic messages                                                                           |
| <code>--mfc</code>                            | Enables multi-file compilation                                                                                                   |
| <code>--minimize_constant_tables</code>       | Minimizes the use of constant tables                                                                                             |
| <code>--misrac1998</code>                     | Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .           |
| <code>--misrac2004</code>                     | Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .           |
| <code>--misrac_verbose</code>                 | <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> or the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> . |
| <code>--module_name</code>                    | Sets the object module name                                                                                                      |
| <code>--no_clustering</code>                  | Disables static clustering optimizations                                                                                         |
| <code>--no_code_motion</code>                 | Disables code motion optimization                                                                                                |
| <code>--no_cse</code>                         | Disables common subexpression elimination                                                                                        |
| <code>--no_inline</code>                      | Disables function inlining                                                                                                       |
| <code>--no_path_in_file_macros</code>         | Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>                       |
| <code>--no_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                                                                           |

Table 32: Compiler options summary (Continued)

| Command line option                       | Description                                                            |
|-------------------------------------------|------------------------------------------------------------------------|
| <code>--no_tbaa</code>                    | Disables type-based alias analysis                                     |
| <code>--no_typedefs_in_diagnostics</code> | Disables the use of typedef names in diagnostics                       |
| <code>--no_unroll</code>                  | Disables loop unrolling                                                |
| <code>--no_warnings</code>                | Disables all warnings                                                  |
| <code>--no_wrap_diagnostics</code>        | Disables wrapping of diagnostic messages                               |
| <code>-O</code>                           | Sets the optimization level                                            |
| <code>-o</code>                           | Sets the object filename. Alias for <code>--output</code> .            |
| <code>--omit_types</code>                 | Excludes type information                                              |
| <code>--only_stdout</code>                | Uses standard output only                                              |
| <code>--output</code>                     | Sets the object filename                                               |
| <code>--pending_instantiations</code>     | Sets the maximum number of instantiations of a given C++ template.     |
| <code>--predef_macros</code>              | Lists the predefined symbols.                                          |
| <code>--preinclude</code>                 | Includes an include file before reading the source file                |
| <code>--preprocess</code>                 | Generates preprocessor output                                          |
| <code>--public_eql</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>--strict</code>                     | Checks for strict compliance with Standard C/C++                       |
| <code>--system_include_dir</code>         | Specifies the path for system include files                            |
| <code>--unaligned_word_access</code>      | Allows unaligned word access                                           |
| <code>--use_cplusplus_inline</code>       | Uses C++ inline semantics in C99                                       |
| <code>--variable_enum_size</code>         | Enables enum size optimization                                         |
| <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 |

Table 32: Compiler options summary (Continued)

| Command line option                      | Description                    |
|------------------------------------------|--------------------------------|
| <code>--warnings_affect_exit_code</code> | Warnings affect exit code      |
| <code>--warnings_are_errors</code>       | Warnings are treated as errors |

Table 32: Compiler options summary (Continued)

## Descriptions of compiler options

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



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

### `--avr32_dsp_instructions`

|             |                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--avr32_dsp_instructions={enabled disabled}</code>                                                                                                                                                                                                                                                                                                      |                                                                                                                         |
| Parameters  | enabled                                                                                                                                                                                                                                                                                                                                                       | Enables the <code>dsp</code> block of instructions. (Default when compiling for the <code>avr32b</code> architecture.)  |
|             | disabled                                                                                                                                                                                                                                                                                                                                                      | Disables the <code>dsp</code> block of instructions. (Default when compiling for the <code>avr32a</code> architecture.) |
| Description | Use this option to enable the <code>dsp</code> block of instructions. This option can be used together with the <code>--core</code> option to control the generated code. By default, <code>dsp</code> instructions are enabled when compiling for the <code>avr32b</code> architecture and disabled when compiling for the <code>avr32a</code> architecture. |                                                                                                                         |
| See also    | <i>Instruction set extensions</i> , page 56                                                                                                                                                                                                                                                                                                                   |                                                                                                                         |



**Project>Options>General Options>Target>Enable DSP instructions**

### `--avr32_flashvault`

|            |                                                    |                                    |
|------------|----------------------------------------------------|------------------------------------|
| Syntax     | <code>--avr32_flashvault={enabled disabled}</code> |                                    |
| Parameters | enabled                                            | Enables secure mode instructions.  |
|            | disabled                                           | Disables secure mode instructions. |

**Description** Use this option to enable the use of secure mode instructions (FlashVault). This option is only relevant when **Project>Options>General Options>Target>Device** is set to either **AVR32A (Generic)** or **AVR32B (Generic)**.



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

## --avr32\_fpu\_instructions

**Syntax** `--avr32_fpu_instructions={enabled|disabled}`

**Parameters**

|                       |                                                      |
|-----------------------|------------------------------------------------------|
| <code>enabled</code>  | Enables the <code>fpu</code> block of instructions.  |
| <code>disabled</code> | Disables the <code>fpu</code> block of instructions. |

**Description** Use this option to enable the `fpu` block of instructions. This option can be used together with the `--core` option to control the generated code. `fpu` instructions are disabled by default.

**See also** *Instruction set extensions*, page 56 and *--fp\_implementation*, page 257



**Project>Options>General Options>Target>Enable FPU instructions**

## --avr32\_rmw\_instructions

**Syntax** `--avr32_rmw_instructions={enabled|disabled}`

**Parameters**

|                       |                                                                                                                         |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------|
| <code>enabled</code>  | Enables the <code>rmw</code> block of instructions. (Default when compiling for the <code>avr32a</code> architecture.)  |
| <code>disabled</code> | Disables the <code>rmw</code> block of instructions. (Default when compiling for the <code>avr32b</code> architecture.) |

**Description** Use this option to enable the `rmw` block of instructions. This option can be used together with the `--core` option to control the generated code. By default, `rmw` instructions are enabled when compiling for the `avr32a` architecture and disabled when compiling for the `avr32b` architecture.

**See also** *Instruction set extensions*, page 56

**Project>Options>General Options>Target>Enable RMW instructions****--avr32\_simd\_instructions**

|             |                                                                                                                                                                                                                                                                                                                                       |                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--avr32_simd_instructions={enabled disabled}</code>                                                                                                                                                                                                                                                                             |                                                                                                             |
| Parameters  | enabled                                                                                                                                                                                                                                                                                                                               | Enables the <code>simd</code> block of instructions. (Default when compiling for the avr32b architecture.)  |
|             | disabled                                                                                                                                                                                                                                                                                                                              | Disables the <code>simd</code> block of instructions. (Default when compiling for the avr32a architecture.) |
| Description | Use this option to enable the <code>simd</code> block of instructions. This option can be used together with the <code>--core</code> option to control the generated code. By default, <code>simd</code> instructions are enabled when compiling for the avr32b architecture and disabled when compiling for the avr32a architecture. |                                                                                                             |
| See also    | <i>Instruction set extensions</i> , page 56                                                                                                                                                                                                                                                                                           |                                                                                                             |

**Project>Options>General Options>Target>Enable SIMD instructions****--c89**

|             |                                                                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--c89</code>                                                                                                                                |
| 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 187.                                                                                                            |

**Project>Options>C/C++ Compiler>Language 1>C dialect>C89**

## --char\_is\_signed

Syntax `--char_is_signed`

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

**Note:** The runtime library is compiled without the `--char_is_signed` option and cannot be used with code that is compiled with this option.



**Project>Options>C/C++ Compiler>Language 2>Plain ‘char’ is**

## --char\_is\_unsigned

Syntax `--char_is_unsigned`

Description Use this option to make the compiler interpret the plain `char` type as unsigned. This is the default interpretation of the plain `char` type.



**Project>Options>C/C++ Compiler>Language 2>Plain ‘char’ is**

## --code\_model

Syntax `--code_model={small|medium|large}`

Parameters

|                              |                                                                                     |
|------------------------------|-------------------------------------------------------------------------------------|
| <code>small</code> (default) | Allows for up to 1 Mbyte of memory                                                  |
| <code>medium</code>          | Allows for up to 1 Mbyte of memory which can be placed anywhere in the memory range |
| <code>large</code>           | Allows for up to 4 Gbytes of memory                                                 |

Description Use this option to select the code model, which means a default placement of functions. 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 71.



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

**--core**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                            |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| Syntax      | <code>--core={avr32a avr32b}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                            |
| Parameters  | <code>avr32a</code> (default)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | Generates code for the AVR32A architecture |
|             | <code>avr32b</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Generates code for the AVR32B architecture |
| Description | <p>Use this option to select the processor architecture for which the code will be generated. If you do not use the option to specify a core, the compiler generates code for the <code>avr32b</code> architecture by default. Note that all modules of your application must use the same core.</p> <p>The compiler supports the different AVR32 microprocessor architectures and derivatives based on these architectures. The object code that the compiler generates for the different architectures is binary compatible. However, it is not possible to link <code>avr32a</code> code with <code>avr32b</code> code because of the different values of the runtime attribute.</p> |                                            |



**Project>Options>General Options>Target>Device**

**--core\_revision**

|             |                                                                                                                                                                                 |                                                                                                                                        |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--core_revision revision</code>                                                                                                                                           |                                                                                                                                        |
| Parameters  | <code>revision</code>                                                                                                                                                           | The revision to use. Valid numbers are 1 and up. Refer to the hardware documentation to determine which core revision your device has. |
| Description | <p>Use this option to generate code for the specified revision of the instruction set architecture. This option cannot be used together with the <code>--cpu</code> option.</p> |                                                                                                                                        |



**Project>Options>General Options>Target>Core revision**

**--cpu**

|            |                           |                             |
|------------|---------------------------|-----------------------------|
| Syntax     | <code>--cpu=device</code> |                             |
| Parameters | <code>device</code>       | Specifies a specific device |

**Description** The compiler supports different devices, alternatively referred to as *parts*. Use this option to select a specific device for which the code will be generated. If this option is not specified, a generic AVR32A device is assumed.



**Project>Options>General Options>Target>Device**

## --cpu\_info

**Syntax** `--cpu_info file`

**Parameters**

*file* The path and filename of the file containing the configuration information for the device

**Description** Use this option to read a file containing configuration information about a specific device. The compiler can then fine-tune the code generation for that target. The `--cpu_info` option is only required when new devices are added in between releases of IAR Embedded Workbench for AVR32. The compiler contains a database of all devices known at release time.

**See also** `--cpu`, page 247



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

## -D

**Syntax** `-D symbol[=value]`

**Parameters**

*symbol* The name of the preprocessor symbol

*value* The value of the preprocessor symbol

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

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

`-Dsymbol`



is equivalent to:

```
#define symbol 1
```

To get the equivalence of:

```
#define FOO
```

specify the = sign but nothing after, for example:

```
-DFOO=
```



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

## --data\_model

Syntax

```
--data_model={small|large}
```

Parameters

`small` (default in small code model)      Accesses the lower and upper 1 Mbyte of memory

`large` (default in the large code model)      Accesses the entire 4 Gbytes 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

*Data models*, page 66.



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

## --debug, -r

Syntax

```
--debug  
-r
```

Description

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

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



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

## --dependencies

|            |                                                                    |                                                                                           |
|------------|--------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| Syntax     | <code>--dependencies [= [i m n] [s]] {filename directory +}</code> |                                                                                           |
| Parameters | <code>i</code> (default)                                           | Lists only the names of files                                                             |
|            | <code>m</code>                                                     | Lists in makefile style (multiple rules)                                                  |
|            | <code>n</code>                                                     | Lists in makefile style (one rule)                                                        |
|            | <code>s</code>                                                     | Suppresses system files                                                                   |
|            | <code>+</code>                                                     | Gives the same output as <code>-o</code> , but with the filename extension <code>d</code> |

See also *Rules for specifying a filename or directory as parameters*, page 238.

**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.r82: c:\iar\product\include\stdio.h
foo.r82: 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:

```
%.r82 : %.c
        $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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



This option is not available in the IDE.

## --diag\_error

Syntax

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

Parameters

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

Description

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



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

## --diag\_remark

Syntax

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

Parameters

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

Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line.

**Note:** By default, remarks are not displayed; use the `--remarks` option to display them.



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

## --diag\_suppress

Syntax

```
--diag_suppress=tag[, tag, ...]
```

Parameters

*tag*

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

Description

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



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

## --diag\_warning

Syntax

--diag\_warning=*tag*[, *tag*, ...]

Parameters

*tag*

The number of a diagnostic message, for example the message number Pe826

Description

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



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

## --diagnostics\_tables

Syntax

--diagnostics\_tables {*filename*|*directory*}

Parameters

See *Rules for specifying a filename or directory as parameters*, page 238.

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.

## --disable\_inline\_asm\_label\_replacement

Syntax `--disable_inline_asm_label_replacement`

Description Use this option to disable label replacement in inline assembler statements. Disabling label replacement will make labels shared between inline assembler statements within a function, which means that one inline assembler statement can refer to a label in another inline assembler statement.

**Note:** The compiler leaves no guarantees on the content of any register nor of the layout of the stack frame upon entry of an inline assembler statement, unless you explicitly request it through the use of inline assembler. Branching between two inline assembler statements results in *undefined behavior*.



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

## --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 260 and *Multi-file compilation units*, page 219.



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

## --dlib\_config

Syntax `--dlib_config filename.h|config`

Parameters

*filename* A DLIB configuration header file, see *Rules for specifying a filename or directory as parameters*, page 238.

*config*

The default configuration file for the specified configuration will be used. Choose between:

*none*, no configuration will be used

*normal*, the normal library configuration will be used (default)

*full*, the full library configuration will be used.

**Description**

Use this option to specify which library configuration to use, either by specifying an explicit file or by specifying a library configuration—in which case the default file for that library configuration will be used. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `avr32\lib`. For examples and information about prebuilt runtime libraries, see *Using prebuilt libraries*, page 121.

If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see *Building and using a customized library*, page 130.



To set related options, choose:

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

**-e**

**Syntax**

`-e`

**Description**

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

**Note:** The `-e` option and the `--strict` option cannot be used at the same time.

**See also**


*Enabling language extensions*, page 189.




**Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions**

**Note:** By default, this option is selected in the IDE.


**--ec++**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --ec++                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Description | <p>In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.</p> <p> <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 1&gt;C++</b><br/>and<br/><b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 1&gt;C++ dialect&gt;Embedded C++</b></p> |

**--eec++**

|             |                                                                                                                                                                                                                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --eec++                                                                                                                                                                                                                                                                                                                |
| Description | <p>In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.</p>                                              |
| See also    | <p><i>Extended Embedded C++</i>, page 196.</p> <p> <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 1&gt;C++</b><br/>and<br/><b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 1&gt;C++ dialect&gt;Extended Embedded C++</b></p> |

**--enable\_multibytes**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --enable_multibytes                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Description | <p>By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.</p> <p>Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.</p> <p> <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language 2&gt;Enable multibyte support</b></p> |

## --enable\_restrict

Syntax `--enable_restrict`

Description Enables the Standard C keyword `restrict`. This option can be useful for improving analysis precision during optimization.



To set this option, use **Project>Options>C/C++ Compiler>Extra options**

## --error\_limit

Syntax `--error_limit=n`

Parameters

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

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



This option is not available in the IDE.

## -f

Syntax `-f filename`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 238.

Description

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

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


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




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



## --fp\_implementation

|             |                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--fp_implementation={default small fast fpu}</code>                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                |
| Parameters  | default                                                                                                                                                                                                                                                                                                                                                           | The default alternative depends on the used optimization level. High-speed optimization uses the fast implementation while all other optimization levels use the small implementation.<br><br>If the option <code>--avr32_fpu_instructions</code> is enabled, <code>fpu</code> is the default. |
|             | small                                                                                                                                                                                                                                                                                                                                                             | Uses the small floating-point implementation.                                                                                                                                                                                                                                                  |
|             | fast                                                                                                                                                                                                                                                                                                                                                              | Uses the fast floating-point implementation.                                                                                                                                                                                                                                                   |
|             | fpu                                                                                                                                                                                                                                                                                                                                                               | Uses <code>fpu</code> instructions for floating-point values.                                                                                                                                                                                                                                  |
| Description | Use this option to select the floating-point implementation to use. Only one floating-point implementation will be linked into the final application, with the fast implementation taking precedence. This means that if at least one module that uses the fast implementation is linked into the application, then all modules will use the fast implementation. |                                                                                                                                                                                                                                                                                                |
|             |                                                                                                                                                                                                                                                                                  | <b>Project&gt;Options&gt;C/C++ Compiler&gt;Optimizations&gt;Floating-point implementation</b>                                                                                                                                                                                                  |

## --guard\_calls

|             |                                                                                                                                         |                                                                                        |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--guard_calls</code>                                                                                                              |                                                                                        |
| Description | Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment. |                                                                                        |
| See also    | <i>Managing a multithreaded environment</i> , page 149.                                                                                 |                                                                                        |
|             |                                                      | To set this option, use <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b> . |

## --header\_context

|        |                               |
|--------|-------------------------------|
| Syntax | <code>--header_context</code> |
|--------|-------------------------------|

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



This option is not available in the IDE.

## -I

**Syntax** `-I path`

**Parameters** `path` The search path for `#include` files

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

**See also** *Include file search procedure*, page 232.



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

## -l

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

**Parameters**

|             |                                                                                                                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a (default) | Assembler list file                                                                                                                                                                                                                                     |
| A           | Assembler list file with C or C++ source as comments                                                                                                                                                                                                    |
| b           | Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included * |
| B           | Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included * |
| c           | C or C++ list file                                                                                                                                                                                                                                      |

|             |                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------|
| C (default) | C or C++ list file with assembler source as comments                                                                               |
| D           | C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values                  |
| N           | No diagnostics in file                                                                                                             |
| H           | Include source lines from header files in output. Without this option, only source lines from the primary source file are included |

\* This makes the list file less useful as input to the assembler, but more useful for reading by a human.

See also *Rules for specifying a filename or directory as parameters*, page 238.

**Description** Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.



To set related options, choose:

**Project>Options>C/C++ Compiler>List**

## --library\_module

**Syntax** --library\_module

**Description** Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.



**Project>Options>C/C++ Compiler>Output>Module type>Library Module**

## --macro\_positions\_in\_diagnostics

**Syntax** --macro\_positions\_in\_diagnostics

**Description** Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.



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

## --mfc

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --mfc                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Description | Use this option to enable <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>iccavr32 myfile1.c myfile2.c myfile3.c --mfc</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| See also    | <code>--discard_unused_publics</code> , page 253, <code>--output</code> , <code>-o</code> , page 266, and <i>Multi-file compilation units</i> , page 219.                                                                                                                                                                                                                                                                                                                                                                       |



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

## --minimize\_constant\_tables

|             |                                                         |                                                                                        |
|-------------|---------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--minimize_constant_tables={yes no}</code>        |                                                                                        |
| Parameters  | <code>yes</code>                                        | The compiler tries to avoid constant table entries. (Default for the AVR32A core)      |
|             | <code>no</code>                                         | The compiler generates constant table entries as needed. (Default for the AVR32B core) |
| Description | Use this option to minimize the use of constant tables. |                                                                                        |



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

## --module\_name

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

**Description** Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.

This option is useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.



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

## **--no\_clustering**

**Syntax** `--no_clustering`

**Description** Use this option to disable static clustering optimizations.

**Note:** This option has no effect at optimization.

**See also** *Static clustering*, page 223.



**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 222.



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

## **--no\_cse**

**Syntax** `--no_cse`

**Description** Use this option to disable common subexpression elimination.

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

See also *Common subexpression elimination*, page 221.



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

## **--no\_inline**

Syntax `--no_inline`

Description Use this option to disable function inlining.

See also *Inlining functions*, page 87.



**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 348.



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.

See also *Instruction scheduling*, page 223.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling**

## --no\_size\_constraints

|             |                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_size_constraints</code>                                                                                                                                                          |
| Description | Use this option to relax the normal restrictions for code size expansion when optimizing for high speed.<br><br><b>Note:</b> This option has no effect unless used with <code>-Ohs</code> . |
| See also    | <i>Speed versus size</i> , page 221.                                                                                                                                                        |



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>No size constraints**

## --no\_static\_destruction

|             |                                                                                                                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_static_destruction</code>                                                                                                                                                                                 |
| Description | Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.<br><br>Use this option to suppress the emission of such code. |
| See also    | <i>System termination</i> , page 134.                                                                                                                                                                                |



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

## --no\_system\_include

|             |                                                                                                                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_system_include</code>                                                                                                                                                                                                                        |
| 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 <code>-I</code> compiler option. |
| See also    | <code>--dlib_config</code> , page 253, and <code>--system_include_dir</code> , page 271.                                                                                                                                                                |



**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 222.



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

## --no\_typedefs\_in\_diagnostics

Syntax `--no_typedefs_in_diagnostics`

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

Example 

```
typedef int (*MyPtr)(char const *);
MyPtr p = "My text string";
```

will give an error message like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

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



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

## --no\_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 221.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling**

## --no\_warnings

Syntax `--no_warnings`

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



This option is not available in the IDE.

## --no\_wrap\_diagnostics

Syntax `--no_wrap_diagnostics`

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



This option is not available in the IDE.

## -O

Syntax `-O[n|l|m|h|hs|hz]`

Parameters

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

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

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

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

**See also** *Controlling compiler optimizations*, page 218.



**Project>Options>C/C++ Compiler>Optimizations**

## **--omit\_types**

**Syntax** `--omit_types`

**Description** By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.



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

## **--only\_stdout**

**Syntax** `--only_stdout`

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



This option is not available in the IDE.

## **--output, -o**

**Syntax** `--output {filename|directory}`  
`-o {filename|directory}`

**Parameters** See *Rules for specifying a filename or directory as parameters*, page 238.

**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 `.o`. Use this option to explicitly specify a different output filename for the object code output.



This option is not available in the IDE.

## --pending\_instantiations

**Syntax** `--pending_instantiations number`

**Parameters** `number` An integer that specifies the limit, where 64 is default. If 0 is used, there is no limit.

**Description** Use this option to specify the maximum number of instantiations of a given C++ template that is allowed to be in process of being instantiated at a given time. This is used for detecting recursive instantiations.



**Project>Options>C/C++ Compiler>Extra Options**

## --predef\_macros

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

**Parameters** See *Rules for specifying a filename or directory as parameters*, page 238.

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

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

Note that this option requires that you specify a source file on the command line.



This option is not available in the IDE.

## --preinclude

|             |                                                                                                                                                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--preinclude <i>includefile</i></code>                                                                                                                                                                                                                  |
| Parameters  | See <i>Rules for specifying a filename or directory as parameters</i> , page 238.                                                                                                                                                                             |
| Description | Use this option to make the compiler read the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol. |



**Project>Options>C/C++ Compiler>Preprocessor>Preinclude file**

## --preprocess

|             |                                                                                                                                                                                                                                                                        |   |                   |   |                 |   |                           |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|-------------------|---|-----------------|---|---------------------------|
| Syntax      | <code>--preprocess [= [c] [n] [l]] {<i>filename</i> <i>directory</i>}</code>                                                                                                                                                                                           |   |                   |   |                 |   |                           |
| Parameters  | <table> <tr> <td>c</td> <td>Preserve comments</td> </tr> <tr> <td>n</td> <td>Preprocess only</td> </tr> <tr> <td>l</td> <td>Generate #line directives</td> </tr> </table> <p>See also <i>Rules for specifying a filename or directory as parameters</i>, page 238.</p> | c | Preserve comments | n | Preprocess only | l | Generate #line directives |
| c           | Preserve comments                                                                                                                                                                                                                                                      |   |                   |   |                 |   |                           |
| n           | Preprocess only                                                                                                                                                                                                                                                        |   |                   |   |                 |   |                           |
| l           | Generate #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\_equ

|               |                                                                                                                                                                                                        |               |                                                |              |                                                   |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|------------------------------------------------|--------------|---------------------------------------------------|
| Syntax        | <code>--public_equ <i>symbol</i> [= <i>value</i>]</code>                                                                                                                                               |               |                                                |              |                                                   |
| Parameters    | <table> <tr> <td><i>symbol</i></td> <td>The name of the assembler symbol to be defined</td> </tr> <tr> <td><i>value</i></td> <td>An optional value of the defined assembler symbol</td> </tr> </table> | <i>symbol</i> | The name of the assembler symbol to be defined | <i>value</i> | An optional value of the defined assembler symbol |
| <i>symbol</i> | The name of the assembler symbol to be defined                                                                                                                                                         |               |                                                |              |                                                   |
| <i>value</i>  | An optional value of the defined assembler symbol                                                                                                                                                      |               |                                                |              |                                                   |
| Description   | This option is equivalent to defining a label in assembler language using the EQU directive and exporting it using the PUBLIC directive. This option can be used more than once on the command line.   |               |                                                |              |                                                   |



This option is not available in the IDE.

## --relaxed\_fp

Syntax

```
--relaxed_fp
```

Description

Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:

- The expression consists of both single- and double-precision values
- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

Note that performing the calculation in single precision instead of double precision might cause a loss of accuracy.

Example

```
float F(float a, float b)
{
    return a + b * 3.0;
}
```

The C standard states that `3.0` in this example has the type `double` and therefore the whole expression should be evaluated in `double` precision. However, when the `--relaxed_fp` option is used, `3.0` will be converted to `float` and the whole expression can be evaluated in `float` precision.



To set related options, choose:

**Project>Options>C/C++ Compiler>Language 2>Floating-point semantics**

## --remarks

Syntax

```
--remarks
```

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also

*Severity levels*, page 235.



**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.

## **--strict**

Syntax

`--strict`

Description

By default, the compiler accepts a relaxed superset of Standard C and C++. Use this option to ensure that the source code of your application instead conforms to strict Standard C and C++.

**Note:** The `-e` option and the `--strict` option cannot be used at the same time.

See also

*Enabling language extensions*, page 189.



**Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict**

## --system\_include\_dir

Syntax

`--system_include_dir path`

Parameters

*path* The path to the system include files, see *Rules for specifying a filename or directory as parameters*, page 238.

Description

By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

See also

`--dlib_config`, page 253, and `--no_system_include`, page 263.



This option is not available in the IDE.

## --unaligned\_word\_access

Syntax

`--unaligned_word_access={enabled|disabled}`

Parameters

*enabled* Allows the compiler to use unaligned word accesses.  
*disabled (default)* Prohibits the compiler to use unaligned word accesses.

Description

By default, the compiler uses hard alignment, in other words, misaligned accesses do not work. Use this option to make the compiler use the `ld.w` and `st.w` instructions for unaligned word accesses. This will increase the speed and reduce the size of applications that uses packed structures. When this option is used, the calling convention used by the compiler is affected for function calls; 4- and 8-byte aggregate objects with alignment less than four bytes can be passed in registers.

See also

*Function entrance*, page 173 and for information about alignment control, see *Extensions for embedded systems programming*, page 190.



**Project>Options>C/C++ Compiler>Optimizations>Allow unaligned word accesses**

## --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 87                                                                                                                             |



**Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics**

## --variable\_enum\_size

|             |                                                                |                                                                                                                        |
|-------------|----------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--variable_enum_size={enabled disabled}</code>           |                                                                                                                        |
| Parameters  | <code>enabled</code>                                           | Enables <code>enum</code> size optimization. <code>enum</code> types are represented using the smallest possible type. |
|             | <code>disabled</code>                                          | Disables <code>enum</code> size optimization. <code>enum</code> types are represented using 32-bit types or larger.    |
| Description | Use this option to enable <code>enum</code> size optimization. |                                                                                                                        |
| See also    | <i>The enum type</i> , page 276                                |                                                                                                                        |



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

## --vla

|             |                                                                                                                                                                                                                                                                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--vla</code>                                                                                                                                                                                                                                                                                                                                                      |
| Description | Use this option to enable support for C99 variable length arrays. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the <code>--c89</code> compiler option.<br><br><b>Note:</b> <code>--vla</code> 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 187.                                                                                                                                                                                                                                                                                                                                  |





**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 252.



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



# Data representation

- Alignment
- Basic data types—integer types
- Basic data types—floating-point types
- Pointer types
- Structure types
- Type qualifiers
- Data types in C++

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

---

## Alignment

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

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

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

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

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

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of

the structure. For more information about pad bytes, see *Packed structure types*, page 283.

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

---

## Basic data types—integer types

The compiler supports both all Standard C basic data types and some additional types.

### INTEGER TYPES—AN OVERVIEW

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

| Data type                       | Size    | Range                   | Alignment |
|---------------------------------|---------|-------------------------|-----------|
| <code>bool</code>               | 8 bits  | 0 to 1                  | 1         |
| <code>char</code>               | 8 bits  | 0 to 255                | 1         |
| <code>signed char</code>        | 8 bits  | -128 to 127             | 1         |
| <code>unsigned char</code>      | 8 bits  | 0 to 255                | 1         |
| <code>signed short</code>       | 16 bits | -32768 to 32767         | 2         |
| <code>unsigned short</code>     | 16 bits | 0 to 65535              | 2         |
| <code>signed int</code>         | 32 bits | $-2^{31}$ to $2^{31}-1$ | 4         |
| <code>unsigned int</code>       | 32 bits | 0 to $2^{32}-1$         | 4         |
| <code>signed long</code>        | 32 bits | $-2^{31}$ to $2^{31}-1$ | 4         |
| <code>unsigned long</code>      | 32 bits | 0 to $2^{32}-1$         | 4         |
| <code>signed long long</code>   | 64 bits | $-2^{63}$ to $2^{63}-1$ | 4         |
| <code>unsigned long long</code> | 64 bits | 0 to $2^{64}-1$         | 4         |

*Table 33: Integer types*

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

### BOOL

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

### THE ENUM TYPE

The compiler will always use the type `long` to hold enum constants, preferring `signed` rather than `unsigned`, unless the `--variable_enum_size` option has been enabled.

## THE CHAR TYPE

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

## THE WCHAR\_T TYPE

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

## BITFIELDS

In 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 AVR32, plain integer types are treated as signed.

Bitfields in expressions are treated as `int` if `int` can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.

Each bitfield is placed 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.

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.

Use the `#pragma bitfield` directive to choose which bitfield allocation strategy to use, see *bitfields*, page 307.

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

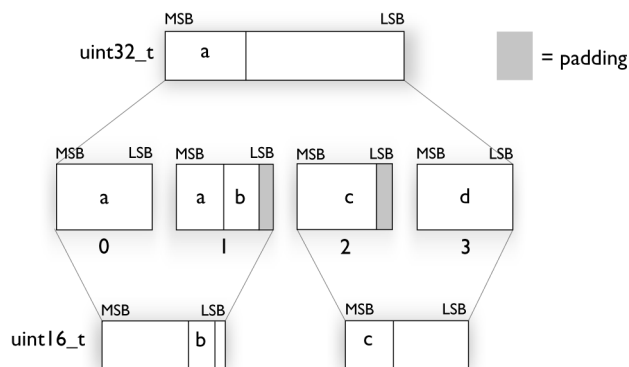
To place the first bitfield, *a*, the compiler allocates a 32-bit container at offset 0 and puts *a* into the first and second bytes of the container.

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.

In each case, each bitfield is allocated starting from the most 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.

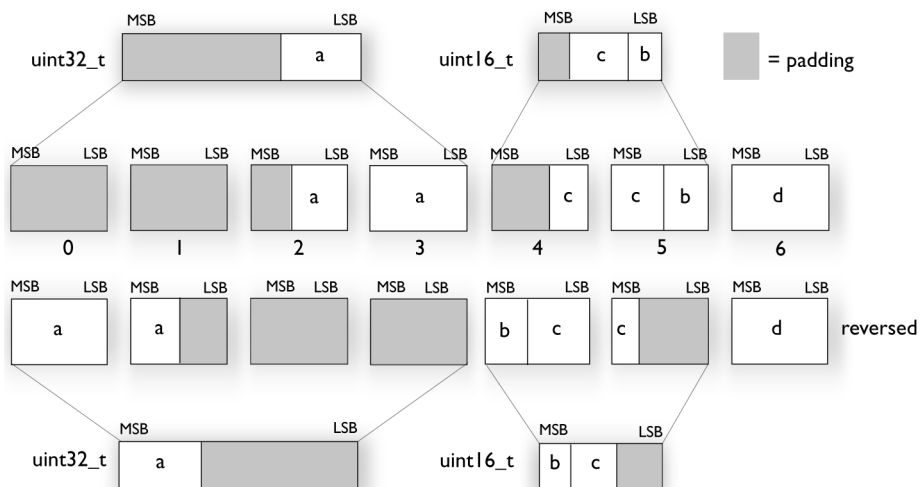
To place the second bitfield, `b`, a new container is allocated at offset 4, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

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` in big-endian mode:



## Basic data types—floating-point types

In the IAR C/C++ Compiler for AVR32, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

| Type                     | Size if double=32 | Size if double=64 |
|--------------------------|-------------------|-------------------|
| <code>float</code>       | 32 bits           | 32 bits           |
| <code>double</code>      | 32 bits           | 64 bits           |
| <code>long double</code> | 32 bits           | 64 bits           |

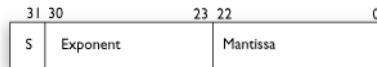
Table 34: Floating-point types

## 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:

$$(-1)^S * 2^{(\text{Exponent}-127)} * 1.\text{Mantissa}$$

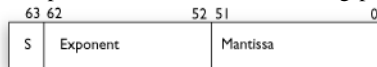
The range of the number is at least:

$$\pm 1.18\text{E}-38 \text{ to } \pm 3.39\text{E}+38$$

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.
- 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, and they can address the entire memory. The internal representation of a function pointer is the actual address it refers to divided by two.

These function pointers are available:

| Keyword               | Address range                            | Pointer size | Alignment |
|-----------------------|------------------------------------------|--------------|-----------|
| <code>__code21</code> | 0x0-0x000FFFF,<br>0xFFFF0000-0xFFFFFFFF* | 32 bits      | 4         |
| <code>__code32</code> | 0x0-0xFFFFFFFF                           | 32 bits      | 4         |

Table 35: Function pointers

\* The upper range is used for SFRs.

### DATA POINTERS

Data pointers have three sizes: 8, 16, or 24 bits. These data pointers are available:

| Keyword               | Pointer size | Index type | Address range                             |
|-----------------------|--------------|------------|-------------------------------------------|
| <code>__data17</code> | 32 bits      | signed int | 0x0-0x0001FFFF,<br>0xFFFFE0000-0xFFFFFFFF |
| <code>__data21</code> | 32 bits      | signed int | 0x0-0x000FFFFF,<br>0xFFFF00000-0xFFFFFFFF |
| <code>__data32</code> | 32 bits      | signed int | 0x0-0xFFFFFFFF                            |

Table 36: Data pointers

**Note:** You cannot create pointers to variables declared with the `__sysreg` or the `__dbgreg` keyword.

## 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 unsigned integer type to a pointer of a larger
- Casting a value of a signed integer type to a pointer of a larger type is performed by sign extension
- Casting a pointer type to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result

### size\_t

`size_t` is the unsigned integer type of the result of the `sizeof` operator. In the IAR C/C++ Compiler for AVR32, 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 AVR32, the type used for `ptrdiff_t` is the signed integer variant of the `size_t` type.

### intptr\_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for AVR32, the type used for `intptr_t` is `signed int`.

### uintptr\_t

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

---

## Structure types

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

## ALIGNMENT OF STRUCTURE TYPES

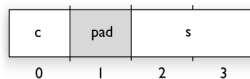
The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. Note that this alignment requirement also applies to a member that is a structure. To allow arrays of aligned structure objects, the size of a `struct` is adjusted to an even multiple of the alignment.

## GENERAL LAYOUT

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

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

This diagram shows the layout in memory:



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

## PACKED STRUCTURE TYPES

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

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

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

This example declares a packed structure:

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

```
#pragma pack()
```

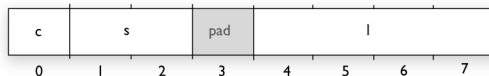
The structure `S` has this memory layout:



The next example declares a new non-packed structure, `S2`, that contains the structure `S` declared in the previous example:

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

The structure `S2` has this memory layout



The structure `S` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 4, which means that alignment of the structure `S2` will become 4.

For more information, see *Alignment of elements in a structure*, page 212.

---

## Type qualifiers

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

### DECLARING OBJECTS VOLATILE

By declaring an object `volatile`, the compiler is informed that the value of the object can change beyond the compiler's control. The compiler must also assume that any

accesses can have side effects—thus all accesses to the `volatile` object must be preserved.

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

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

### Definition of access to volatile objects

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

- The compiler considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type
- Adding a `const` qualifier to a `volatile` object will make write accesses to the object impossible. However, the object will be placed in RAM as specified by the C standard.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for AVR32 are described below.

### Rules for accesses

In the IAR C/C++ Compiler for AVR32, 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 these combinations of memory types and data types:

| Memory type                                   | Data type                 | Comment                                                                                  |
|-----------------------------------------------|---------------------------|------------------------------------------------------------------------------------------|
| <code>__data17</code>                         | 8-, 16-, and 32-bit types | RMW instructions may be used for certain bit accesses and single bit bitfield operations |
| <code>__data21</code> , <code>__data32</code> | 8-, 16-, and 32-bit types |                                                                                          |
| <code>__dbgreg</code> , <code>__sysreg</code> | 32-bit types              |                                                                                          |

Table 37: Volatile accesses

For all combinations of object types not listed, 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, follow this example:

```
/* Header */
extern int const xVar;
#define x (*(int const volatile *) &xVar)

/* Source that uses x */
int DoSomething()
{
    return x;
}

/* Source that defines x */
#pragma segment = "FLASH"
int const xVar @ "FLASH" = 6;
```

The segment `FLASH` contains the initializers. They must be flashed manually when the application starts up.

Thereafter, the initializers can be reflashed with other values at any time.

## DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities

to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

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

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

---

## Data types in C++

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





# Extended keywords

- General syntax rules for extended keywords
- Summary of extended keywords
- Descriptions of extended keywords

---

## General syntax rules for extended keywords

The compiler provides a set of attributes that can be used on functions or data objects to support specific features of the AVR32 microprocessor. 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 293. 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 254.

### 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 microprocessor.

Available *function memory attributes*:

```
__code21, code32
```

Available *data memory attributes*:

```
__data17, __data21, __data32, __dbgreg, and __sysreg.
```

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):

```
__acall, __exception, __flashvault, __flashvault_impl,
__imported_interrupt, __monitor, __nested, and __scall
```

Available *data type attributes*:

```
__packed, const, volatile
```

You can specify as many type attributes as required for each level of pointer indirection.

## Syntax for type attributes used on data objects

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

```
__data17 int i;
int __data17 j;
```

Both `i` and `j` are placed in `data17` 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 __data17 * p;      /* integer in data17 memory */
int * __data17 p;     /* pointer in data17 memory */
```

```
__data17 int * p;      /* pointer in data17 memory */
```

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 __data17 int d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in data17 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=__data17
int * q2;
```

The variable `q2` is placed in data17 memory.

For more examples of using memory attributes, see *More examples*, page 65.

### Syntax for type attributes used on functions

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

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

or

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

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

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

To declare a function pointer, use this syntax:

```
int (__code21 * fp) (double);
```

After this declaration, the function pointer `fp` points to data17 memory.

An easier way of specifying storage is to use type definitions:

```
typedef __code21 void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

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

## OBJECT ATTRIBUTES

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
```

- Object attributes that can be used for functions and variables:

```
location, @, __root
```

- Object attributes that can be used for functions:

```
__intrinsic, __noreturn, __ramfunc, 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 214. For more information about `vector`, see *vector*, page 328.

### 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>__acall</code>  | Supports application calls        |
| <code>__code21</code> | Controls the storage of functions |

Table 38: Extended keywords summary

| Extended keyword                                              | Description                                                                         |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <code>__code32</code>                                         | Controls the storage of functions                                                   |
| <code>__data17</code>                                         | Controls the storage of data objects                                                |
| <code>__data21</code>                                         | Controls the storage of data objects                                                |
| <code>__data32</code>                                         | Controls the storage of data objects                                                |
| <code>__dbgreg</code>                                         | Controls the storage of data objects                                                |
| <code>__exception</code>                                      | Supports exception handlers                                                         |
| <code>__flashvault</code>                                     | Supports FlashVault handlers                                                        |
| <code>__flashvault_impl</code>                                | Supports FlashVault API. Supplementary to <code>__flashvault</code> .               |
| <code>__imported</code>                                       | Signals that a function is not present in the application being compiled and linked |
| <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>__nested</code>                                         | Supports nested interrupts                                                          |
| <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_init</code>                                        | Places a data object in non-volatile memory                                         |
| <code>__noreturn</code>                                       | Informs the compiler that the function will not return                              |
| <code>__packed</code>                                         | Decreases data type alignment to 1                                                  |
| <code>__ramfunc</code>                                        | Makes a function execute in RAM                                                     |
| <code>__root</code>                                           | Ensures that a function or variable is included in the object code even if unused   |
| <code>__scall</code>                                          | Supports supervisor calls                                                           |
| <code>__sysreg</code>                                         | Controls the storage of data objects                                                |

Table 38: Extended keywords summary (Continued)

## Descriptions of extended keywords

This section gives detailed information about each extended keyword.

### `__acall`

Syntax

See *Syntax for type attributes used on functions*, page 291.

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description  | <p>The <code>__acall</code> keyword makes it possible to use the <code>ACALL</code> instruction to call a function. Such function calls can be more compact than a generic function call, and the generated code can become smaller. An <code>__acall</code> function uses the same calling convention as other functions.</p> <p>The <code>ACALL</code> instruction calls the function through a function pointer table located in the linker segment <code>ACTAB</code>. Each <code>__acall</code> function will occupy one of these entries. The entry to be used is specified by a <code>#pragma vector</code> directive immediately before the function declaration.</p> <p><b>Note:</b> If you do not specify a vector and not use the <code>__imported</code> keyword, the compiler will automatically create an entry in the function pointer table.</p> |
| Vector range | 0x0-0x255                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Example      | <pre>#pragma vector=7 __acall int my_acall_function(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| See also     | <code>__imported</code> , page 298, <i>ACALL functions</i> , page 77, and <i>ACALL jump table</i> , page 111.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## `__code21`

|                     |                                                                                                                                                                                                                                                                                                                  |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on functions</i> , page 291.                                                                                                                                                                                                                                              |
| Description         | The <code>__code21</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in code21 memory. You can also use the <code>__code21</code> attribute to create a pointer explicitly pointing to an object located in the data17 memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFF</li> <li>● Pointer size: 4 bytes</li> </ul>                                                                                                                                                                        |
| Example             | <pre>__code21 void myfunction(void);</pre>                                                                                                                                                                                                                                                                       |
| See also            | <i>Code models and memory attributes for function storage</i> , page 71.                                                                                                                                                                                                                                         |

## `__code32`

|             |                                                                                                                                                                                       |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 291.                                                                                                                   |
| Description | The <code>__code32</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in code32 memory. You can also |

use the `__code32` attribute to create a pointer explicitly pointing to an object located in the `data32` memory.

- Storage information
- Address range: Anywhere in memory
  - Pointer size: 4 bytes

Example `__code32 void myfunction(void);`

See also *Code models and memory attributes for function storage*, page 71.

## **\_\_data17**

Syntax See *Syntax for type attributes used on data objects*, page 290.

Description The `__data17` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in `data17` memory.

- Storage information
- Address range: `0x0-0x0001FFFF`, `0xFFFE0000-0xFFFFFFFF`
  - Pointer size: 4 bytes.

Example `__data17 int x;`

See also *Memory types*, page 60.

## **\_\_data21**

Syntax See *Syntax for type attributes used on data objects*, page 290.

Description The `__data21` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in `data21` memory.

- Storage information
- Address range: `0x0-0x000FFFFF`, `0xFFF00000-0xFFFFFFFF`
  - Pointer size: 4 bytes.

Example `__data21 int x;`

See also *Memory types*, page 60.

## **\_\_data32**

|                     |                                                                                                                                                                                          |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on data objects</i> , page 290.                                                                                                                   |
| Description         | The <code>__data32</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in data32 memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0x0–0xFFFFFFFF</li> <li>● Pointer size: 4 bytes.</li> </ul>                                                                      |
| Example             | <code>__data32 int x;</code>                                                                                                                                                             |
| See also            | <i>Memory types</i> , page 60.                                                                                                                                                           |

## **\_\_dbgreg**

|                     |                                                                                                                                                                                                                                                                                        |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on data objects</i> , page 290.                                                                                                                                                                                                                 |
| Description         | The <code>__dbgreg</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in the debug register file. It is not possible to create pointers to variables declared <code>__dbgreg</code> . |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0x0–0x3FC</li> <li>● Maximum object size: 4 bytes</li> <li>● Pointer size: N/A.</li> </ul>                                                                                                                                     |
| Example             | <code>__dbgreg int x;</code>                                                                                                                                                                                                                                                           |
| See also            | <i>Memory types</i> , page 60.                                                                                                                                                                                                                                                         |

## **\_\_exception**

|             |                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 291.                                                                                                                                                                                                                                                                                                       |
| Description | <p>Use the <code>__exception</code> keyword to implement handlers for the exceptions that can occur.</p> <p>To associate the exception handler with a specific exception entry point, use the <code>#pragma exception</code> directive immediately before the function declaration. Each exception handler must be connected to at least one exception handler entry.</p> |



**Example** `__exception int my_exception(void);`

**See also** *Exception handlers*, page 75 and *exception*, page 314. For information about the exception handler offsets, see the AVR32 architecture documentation, supplied by Atmel® Corporation.

## **\_\_flashvault**

**Syntax** See *Syntax for type attributes used on functions*, page 291.

**Description** A defined function declared with the keyword `__flashvault` returns using the `retss` instruction. This allows a secure-mode handler to be implemented in C.

A function declared with the keyword `__flashvault` *and* an associated `#pragma vector` is called using the `sscall` instruction. The compiler automatically initializes R8 with the vector number of the function before the call. No other restrictions or assumptions apply to the function. The secure mode handler can then use R8 to choose between multiple functions. This allows you to create an API file which then can be used when developing an application.

**See also** `__flashvault_impl`, page 297

## **\_\_flashvault\_impl**

**Syntax** See *Syntax for type attributes used on functions*, page 291.

**Description** Use the `__flashvault_impl` keyword when implementing a secure code API. This keyword adjusts the calling convention of a function so that a secure mode API handler can be added between the caller and the called function.

A defined function declared with this keyword is called using normal call instructions and returns using the `ret` instruction.

**Note:** This keyword affects the calling convention, see *Calling convention*, page 170.

**Example**

```
#ifdef FLASHVAULT_PROJECT
#define FLASHVAULT __flashvault_impl
#else
#define FLASHVAULT __flashvault
#endif
#pragma vector=1
uint64_t FLASHVAULT decode_DES_CBC(uint64_t key, uint64_t data);
```

**See also** `__flashvault`, page 297

## **\_\_imported**

|             |                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 291.                                                                                                                                                                                                                                               |
| Description | Use the <code>__imported</code> keyword to signal that the declared function is not present in the application being compiled and linked. It is up to the target operating system or target hardware to make sure that the function is available and present in the table pointed at by the ACBA system register. |
| Example     | <pre>__imported __acall int my_function(void);</pre>                                                                                                                                                                                                                                                              |
| See also    | <i>ACALL functions</i> , page 77, <i>ACALL jump table</i> , page 111, <i>__acall</i> , page 293, and <i>vector</i> , page 328.                                                                                                                                                                                    |

## **\_\_interrupt**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 291.                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description | The <code>__interrupt</code> keyword specifies interrupt functions. To specify one or several interrupt vectors, use the <code>#pragma handler</code> directive. The range of the interrupt groups depends on the device used. It is possible to define an interrupt function using the <code>#pragma handler</code> directive, but then the compiler will not generate an entry in the interrupt vector table.<br><br>An interrupt function must have a <code>void</code> return type and cannot have any parameters. |
| Example     | <pre>#pragma vector=44,3 //group 44 with interrupt level 3 __interrupt void my_interrupt_handler(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                          |
| See also    | <i>Interrupt functions</i> , page 73.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

## **\_\_intrinsic**

|             |                                                                                  |
|-------------|----------------------------------------------------------------------------------|
| Description | The <code>__intrinsic</code> keyword is reserved for compiler internal use only. |
|-------------|----------------------------------------------------------------------------------|

## **\_\_monitor**

|             |                                                                                                                                                                            |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 291.                                                                                                        |
| Description | The <code>__monitor</code> 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 79. For information about related intrinsic functions, see *\_\_disable\_interrupt*, page 337, *\_\_enable\_interrupt*, page 337, *\_\_get\_interrupt\_state*, page 338, and *\_\_set\_interrupt\_state*, page 341, respectively.

## **\_\_nested**

**Syntax** See *Syntax for type attributes used on functions*, page 291.

**Description** Use the `__nested` keyword to implement a nested interrupt, in other words, an interrupt that can be called multiple times.

A nested interrupt saves the interrupt level, the return address register, and the return status register at function entry, and restores these registers at function exit. The application can then lower the interrupt level to allow other interrupts to trigger.

**Example** `__nested __interrupt void my_interrupt_routine(void);`

**See also** *\_\_interrupt*, page 298. For information about interrupt levels and the interrupt controller, see the AVR32 architecture documentation, supplied by Atmel@ Corporation.

## **\_\_no\_alloc, \_\_no\_alloc16**

**Syntax** See *Syntax for object attributes*, page 292.

**Description** Use the `__no_alloc` or `__no_alloc16` object attribute on a constant to make the constant available in the executable file without occupying any space in the linked application.

You cannot access the contents of such a constant from your application. You can take its address, which is an integer offset to the segment of the constant. The type of the offset is `unsigned long` when `__no_alloc` is used, and `unsigned short` when `__no_alloc16` is used.

**Example** `__no_alloc const struct MyData my_data @ "XXX" = {...};`

**See also** *\_\_no\_alloc\_str*, *\_\_no\_alloc\_str16*, page 300.

## **\_\_no\_alloc\_str, \_\_no\_alloc\_str16**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>__no_alloc_str(<i>string_literal</i> @ <i>segment</i>) and __no_alloc_str16(<i>string_literal</i> @ <i>segment</i>) where  <i>string_literal</i>      The string literal that you want to make available in the                       executable file.  <i>segment</i>             The name of the segment to place the string literal in.</pre>                                                                                                                                                      |
| Description | <p>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.</p> <p>The value of the expression is the offset of the string literal in the segment. 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.</p>                                                        |
| 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. Note that there is no such runtime support in C-SPY, unless you use an external plugin module.</p> |
| See also    | <p><code>__no_alloc</code>, <code>__no_alloc16</code>, page 299.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## **\_\_no\_init**

|        |                                                     |
|--------|-----------------------------------------------------|
| Syntax | See <i>Syntax for object attributes</i> , page 292. |
|--------|-----------------------------------------------------|

**Description** Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

**Example**

```
__no_init int myarray[10];
```

**See also** *Non-initialized variables*, page 228.

## **\_\_noreturn**

**Syntax** See *Syntax for object attributes*, page 292.

**Description** The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

**Note:** At optimization levels medium or high, the `__noreturn` keyword might cause incorrect call stack debug information at any point where it can be determined that the current function cannot return.

**Example**

```
__noreturn void terminate(void);
```

## **\_\_packed**

**Syntax** See *Syntax for type attributes used on data objects*, page 290. 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. The type of each member also receives the `__packed` type attribute.  
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.

#### 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 *" -> "int __packed *" not allowed: */
    int * p1 = xp->i;
    /* OK: */
    int __packed * p2 = &xp->i;
    /* OK, char not affected */
    char * p3 = &xp->ch;
}
    
```

See also

*pack*, page 321.

## `__ramfunc`

Syntax

See *Syntax for object attributes*, page 292.

Description

The `__ramfunc` keyword makes a function execute in RAM. Two code segments will be created: one for the RAM execution, and one for the ROM initialization. Functions declared `__ramfunc` are by default stored in the segments named `RAMCODE21` and `RAMCODE32` (depending on the memory type attribute of the function) with the initialization data in the segments `RAMCODE21_ID` and `RAMCODE32_ID`.

Example

```
__ramfunc int FlashPage(char * data, char * page);
```

See also

For more information about `__ramfunc` declared functions in relation to breakpoints, see the *C-SPY® Debugging Guide for AVR32*.

**\_\_root**

|             |                                                                                                                                                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for object attributes</i> , page 292.                                                                                                                                                                                                           |
| Description | A function or variable with the <code>__root</code> 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     | <pre>__root int myarray[10];</pre>                                                                                                                                                                                                                            |
| See also    | For more information about modules, segments, and the link process, see the <i>IAR Linker and Library Tools Reference Guide</i> .                                                                                                                             |

**\_\_scall**

|             |                                                                                                                                                                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | See <i>Syntax for type attributes used on functions</i> , page 291.                                                                                                                                                                                                     |
| Description | Use the <code>__scall</code> keyword to implement a function which will be executed in the supervisor mode of the processor. The function will be called using the <code>SCALL</code> instruction and execution will continue in the supervisor mode exception handler. |
| Example     | <pre>__scall void my_scall(void);</pre>                                                                                                                                                                                                                                 |
| See also    | <i>__exception</i> , page 296 and <i>SCALL functions</i> , page 78. For information about execution modes and the <code>SCALL</code> instruction, see the AVR32 architecture documentation, supplied by Atmel® Corporation.                                             |

**\_\_sysreg**

|                     |                                                                                                                                                                                                                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | See <i>Syntax for type attributes used on data objects</i> , page 290.                                                                                                                                                                                                                  |
| Description         | The <code>__sysreg</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in the system register file. It is not possible to create pointers to variables declared <code>__sysreg</code> . |
| Storage information | <ul style="list-style-type: none"> <li>● Address range: 0x0–0x3FC</li> <li>● Maximum object size: 4 bytes</li> <li>● Pointer size: N/A.</li> </ul>                                                                                                                                      |
| Example             | <pre>__sysreg int x;</pre>                                                                                                                                                                                                                                                              |

See also

*Memory types*, page 60.



# 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>bitfields</code>                   | Controls the order of bitfield members.                                                |
| <code>calls</code>                       | Lists possible called functions for indirect calls.                                    |
| <code>call_graph_root</code>             | Specifies that the function is a call graph root.                                      |
| <code>constseg</code>                    | Places constant variables in a named segment.                                          |
| <code>data_alignment</code>              | Gives a variable a higher (more strict) alignment.                                     |
| <code>dataseg</code>                     | Places variables in a named segment.                                                   |
| <code>default_function_attributes</code> | Sets default type and object attributes for declarations and definitions of functions. |
| <code>default_variable_attributes</code> | Sets default type and object attributes for declarations and definitions of variables. |
| <code>diag_default</code>                | Changes the severity level of diagnostic messages.                                     |
| <code>diag_error</code>                  | Changes the severity level of diagnostic messages.                                     |
| <code>diag_remark</code>                 | Changes the severity level of diagnostic messages.                                     |
| <code>diag_suppress</code>               | Suppresses diagnostic messages.                                                        |
| <code>diag_warning</code>                | Changes the severity level of diagnostic messages.                                     |
| <code>error</code>                       | Signals an error while parsing.                                                        |

*Table 39: Pragma directives summary*

| <b>Pragma directive</b>            | <b>Description</b>                                                                                          |
|------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>exception</code>             | Connects an exception handler with its given exception handler entry                                        |
| <code>flashvault_vector</code>     | Makes the compiler emit a secure mode vector table entry                                                    |
| <code>handler</code>               | Specifies the vector of an interrupt function                                                               |
| <code>include_alias</code>         | Specifies an alias for an include file.                                                                     |
| <code>inline</code>                | Controls inlining of a function.                                                                            |
| <code>language</code>              | Controls the IAR Systems language extensions.                                                               |
| <code>location</code>              | Specifies the absolute address of a variable, or places groups of functions or variables in named segments. |
| <code>message</code>               | Prints a message.                                                                                           |
| <code>object_attribute</code>      | Adds object attributes to the declaration or definition of a variable or function.                          |
| <code>optimize</code>              | Specifies the type and level of an optimization.                                                            |
| <code>pack</code>                  | Specifies the alignment of structures and union members.                                                    |
| <code>__printf_args</code>         | Verifies that a function with a printf-style format string is called with the correct arguments.            |
| <code>public_equ</code>            | Defines a public assembler label and gives it a value.                                                      |
| <code>required</code>              | Ensures that a symbol that is needed by another symbol is included in the linked output.                    |
| <code>rtmodel</code>               | Adds a runtime model attribute to the module.                                                               |
| <code>__scanf_args</code>          | Verifies that a function with a scanf-style format string is called with the correct arguments.             |
| <code>section</code>               | This directive is an alias for <code>#pragma segment</code> .                                               |
| <code>segment</code>               | Declares a segment name to be used by intrinsic functions.                                                  |
| <code>shadow_registers</code>      | Indicates that the register file is shadowed                                                                |
| <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.                    |

*Table 39: Pragma directives summary (Continued)*

| Pragma directive            | Description                                              |
|-----------------------------|----------------------------------------------------------|
| <code>vector</code>         | Specifies the vector of an acall function.               |
| <code>type_attribute</code> | Adds type attributes to a declaration or to definitions. |

Table 39: Pragma directives summary (Continued)

**Note:** For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 398.

## Descriptions of pragma directives

This section gives detailed information about each pragma directive.

### 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 277.     |
|             | <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> .                                                                                                                               |
|             | <code>default</code>                                                                                 | Restores to default layout of bitfield members. The default behavior for the compiler is <code>joined_types</code> .                                                                      |
| 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** *Bitfields*, page 277.

## calls

**Syntax** `#pragma calls=function[, function...]`

**Parameters**

*function*                      Any declared function

**Description** Use this pragma directive to list the functions that can be indirectly called in the following statement. This information can be used for stack usage analysis in the linker.

**Note:** For an accurate result, you must list all possible called functions.

**Example**

```
void Fun1(), Fun2();

void Caller(void (*fp)(void))
{
    #pragma calls = Fun1, Fun2
    (*fp)();
}
```

**See also** *Stack usage analysis*, page 95

## call\_graph\_root

**Syntax** `#pragma call_graph_root[=category]`

**Parameters**

*category*                      A string that identifies an optional call graph root category

**Description** Use this pragma directive to specify that, for stack usage analysis purposes, the immediately following function is a call graph root. You can also specify an optional

category. The compiler will usually automatically assign a call graph root category to interrupt and task functions. If you use the `#pragma call_graph_root` directive on such a function you will override the default category. You can specify any string as a category.

Example `#pragma call_graph_root="interrupt"`

See also *Stack usage analysis*, page 95

## constseg

Syntax `#pragma constseg=[__memoryattribute] {SEGMENT_NAME|default}`

### Parameters

*\_\_memoryattribute* An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.

*SEGMENT\_NAME* A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

default Uses the default segment for constants.

Description Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example 

```
#pragma constseg=__data17 MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

## data\_alignment

Syntax `#pragma data_alignment=expression`

### Parameters

*expression* A constant which must be a power of two (1, 2, 4, etc.).

Description Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

**Note:** Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

## dataseg

|                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                                       |                                                                                                                            |                                  |                                                                                                      |                      |                           |
|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------|----------------------------------|------------------------------------------------------------------------------------------------------|----------------------|---------------------------|
| Syntax                                | <code>#pragma dataseg=[<i>__memoryattribute</i>] {<i>SEGMENT_NAME</i> default}</code>                                                                                                                                                                                                                                                                                                                                                                         |                                       |                                                                                                                            |                                  |                                                                                                      |                      |                           |
| Parameters                            | <table> <tr> <td><code><i>__memoryattribute</i></code></td> <td>An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.</td> </tr> <tr> <td><code><i>SEGMENT_NAME</i></code></td> <td>A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</td> </tr> <tr> <td><code>default</code></td> <td>Uses the default segment.</td> </tr> </table> | <code><i>__memoryattribute</i></code> | An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used. | <code><i>SEGMENT_NAME</i></code> | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker. | <code>default</code> | Uses the default segment. |
| <code><i>__memoryattribute</i></code> | An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.                                                                                                                                                                                                                                                                                                                                    |                                       |                                                                                                                            |                                  |                                                                                                      |                      |                           |
| <code><i>SEGMENT_NAME</i></code>      | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.                                                                                                                                                                                                                                                                                                                                                          |                                       |                                                                                                                            |                                  |                                                                                                      |                      |                           |
| <code>default</code>                  | Uses the default segment.                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                       |                                                                                                                            |                                  |                                                                                                      |                      |                           |
| Description                           | Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared <code>__no_init</code> . The setting remains active until you turn it off again with the <code>#pragma dataseg=default</code> directive.                          |                                       |                                                                                                                            |                                  |                                                                                                      |                      |                           |
| Example                               | <pre>#pragma dataseg=__data17 MY_SECTIONSEGMENT __no_init char myBuffer[1000]; #pragma dataseg=default</pre>                                                                                                                                                                                                                                                                                                                                                  |                                       |                                                                                                                            |                                  |                                                                                                      |                      |                           |

## default\_function\_attributes

|        |                                                                                                                                                                                             |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax | <pre>#pragma default_function_attributes=[ <i>attribute...</i> ]</pre> <p>where <i>attribute</i> can be:</p> <pre><i>type_attribute</i> <i>object_attribute</i> @ <i>segment_name</i></pre> |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Parameters  | <p><i>type_attribute</i>      See <i>Type attributes</i>, page 289.</p> <p><i>object_attribute</i>      See <i>Object attributes</i>, page 292.</p> <p>@ <i>segment_name</i>      See <i>Data and function placement in segments</i>, page 217.</p>                                                                                                                                                                                                                                                                   |
| Description | <p>Use this pragma directive to set default segment placement, type attributes, and object attributes for function declarations and definitions. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.</p> <p>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.</p> |
| Example     | <pre>/* Place following functions in segment MYSEG */ #pragma default_function_attributes = @ "MYSEG" int fun1(int x) { return x + 1; } int fun2(int x) { return x - 1; } /* Stop placing functions into MYSEG */ #pragma default_function_attributes =</pre> <p>has the same effect as:</p> <pre>int fun1(int x) @ "MYSEG" { return x + 1; } int fun2(int x) @ "MYSEG" { return x - 1; }</pre>                                                                                                                       |
| See also    | <p><i>location</i>, page 318</p> <p><i>object_attribute</i>, page 319</p> <p><i>type_attribute</i>, page 327</p>                                                                                                                                                                                                                                                                                                                                                                                                      |

## default\_variable\_attributes

|            |                                                                                                                                                                                     |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <pre>#pragma default_variable_attributes=[ attribute...]</pre> <p>where <i>attribute</i> can be:</p> <pre><i>type_attribute</i> <i>object_attribute</i> @ <i>segment_name</i></pre> |
| Parameters | <p><i>type_attribute</i>      See <i>Type attributes</i>, page 289.</p> <p><i>object_attributes</i>      See <i>Object attributes</i>, page 292.</p>                                |

@ *segment\_name*            See *Data and function placement in segments*, page 217.

**Description**            Use this pragma directive to set default segment placement, type attributes, and object attributes for declarations and definitions of variables with static storage duration. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a `default_variable_attributes` pragma with no attributes restores the initial state of no such defaults being applied to variables with static storage duration.

**Example**

```
/* Place following variables in segment MYSEG */
#pragma default_variable_attributes = @ "MYSEG"
int var1 = 42;
int var2 = 17;
/* Stop placing variables into MYSEG */
#pragma default_variable_attributes =
```

has the same effect as:

```
int var1 @ "MYSEG" = 42;
int var2 @ "MYSEG" = 17;
```

**See also**                *location*, page 318  
*object\_attribute*, page 319  
*type\_attribute*, page 327

## diag\_default

**Syntax**                `#pragma diag_default=tag[, tag, ...]`

**Parameters**            *tag*                    The number of a diagnostic message, for example the message number `Pe177`.

**Description**            Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

**See also**                *Diagnostics*, page 235.



**diag\_error**

|             |                                                                                                             |
|-------------|-------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_error=tag[, tag, ...]</code>                                                             |
| Parameters  | <i>tag</i> The number of a diagnostic message, for example the message number Pe177.                        |
| Description | Use this pragma directive to change the severity level to <code>error</code> for the specified diagnostics. |
| See also    | <i>Diagnostics</i> , page 235.                                                                              |

**diag\_remark**

|             |                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_remark=tag[, tag, ...]</code>                                                                     |
| Parameters  | <i>tag</i> The number of a diagnostic message, for example the message number Pe177.                                 |
| Description | Use this pragma directive to change the severity level to <code>remark</code> for the specified diagnostic messages. |
| See also    | <i>Diagnostics</i> , page 235.                                                                                       |

**diag\_suppress**

|             |                                                                                      |
|-------------|--------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_suppress=tag[, tag, ...]</code>                                   |
| Parameters  | <i>tag</i> The number of a diagnostic message, for example the message number Pe177. |
| Description | Use this pragma directive to suppress the specified diagnostic messages.             |
| See also    | <i>Diagnostics</i> , page 235.                                                       |

## diag\_warning

|             |                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma diag_warning=tag[, tag, ...]</code>                                                                     |
| Parameters  | <i>tag</i> The number of a diagnostic message, for example the message number Pe826.                                  |
| Description | Use this pragma directive to change the severity level to <code>warning</code> for the specified diagnostic messages. |
| See also    | <i>Diagnostics</i> , page 235.                                                                                        |

## error

|             |                                                                                                                                                                                                                                                                                                                                                       |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma error message</code>                                                                                                                                                                                                                                                                                                                    |
| Parameters  | <i>message</i> 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 <code>#error</code> , because the <code>#pragma error</code> directive can be included in a preprocessor macro using the <code>_Pragma</code> form of the directive and only causes an error if the macro is used. |
| Example     | <pre>#if FOO_AVAILABLE #define FOO ... #else #define FOO _Pragma("error\"Foo is not available\"") #endif</pre> <p>If <code>FOO_AVAILABLE</code> is zero, an error will be signaled if the <code>FOO</code> macro is used in actual source code.</p>                                                                                                   |

## exception

|            |                                                                  |
|------------|------------------------------------------------------------------|
| Syntax     | <code>#pragma exception=group, level[, group, level, ...]</code> |
| Parameters | <i>offset</i> The offset from the address in the EVBA register   |

|             |                                                                                                                                                                                                                                                                                     |                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|             | <code>size</code>                                                                                                                                                                                                                                                                   | The size of the exception handler. If the size is set to 0, the size restriction of the handler is disabled, which forces the handler to be generated in the exception table. |
| Description | Use this pragma directive immediately before an exception handler to connect it with one or several exception handler entries. The exception handlers are placed relative to the <code>EVBA</code> register and the offsets are documented in the AVR32 architecture documentation. |                                                                                                                                                                               |
| Example     | <pre>#pragma exception=0x14,4 __exception void my_exception_handler(void) { ... }</pre>                                                                                                                                                                                             |                                                                                                                                                                               |
| See also    | <i>Exception handlers</i> , page 75 and <i>__exception</i> , page 296. For information about the exception handler offsets, see the AVR32 architecture documentation, supplied by Atmel® Corporation.                                                                               |                                                                                                                                                                               |

## flashvault\_vector

|             |                                                                                                                                           |                                                                                                  |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma flashvault_vector={vector, ...}</code>                                                                                      |                                                                                                  |
| Parameters  | <code>vector</code>                                                                                                                       | A FlashVault vector offset. For more information, see the documentation from Atmel® Corporation. |
| Description | A secure mode function defined with this pragma directive makes the compiler emit a secure mode vector table entry at the given location. |                                                                                                  |
| Example     | This example adds a secure mode <code>sscall</code> handler:<br><pre>#pragma flashvault_vector=0x14</pre>                                 |                                                                                                  |

## handler

|            |                                                                |                                                                          |
|------------|----------------------------------------------------------------|--------------------------------------------------------------------------|
| Syntax     | <code>#pragma handler=group, level[, group, level, ...]</code> |                                                                          |
| Parameters | <code>group</code>                                             | The interrupt group to which the handler is connected, which can be 0-63 |

|             |                                                                                                                                                                                                                                                                                                           |                                                                               |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
|             | <i>level</i>                                                                                                                                                                                                                                                                                              | The interrupt level at which the handler should be executed, which can be 0-3 |
| Description | Use this pragma directive immediately before an interrupt function to connect it with its interrupt vector, which is specified by a interrupt group number and an interrupt level. If the interrupt function should handle several different interrupt sources, you can specify several interrupt groups. |                                                                               |
| Example     | <pre>#pragma handler=35,2, 44,3 /*group 35 with interrupt level 2                              group 44 with interrupt level 3 */ __interrupt void my_interrupt_routine1(void) {     ... }  __interrupt void my_interrupt_routine2(void)</pre>                                                            |                                                                               |
| See also    | <i>Interrupt functions</i> , page 73 and <i>__interrupt</i> , page 298.                                                                                                                                                                                                                                   |                                                                               |

## 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 232.                                                                                                                                                                                                                                                                                                                     |                                                                  |

## inline

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                          |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma inline[=forced =never]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                                                                                          |
| 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> <p>Inlining is normally performed only on the High optimization level. Specifying <code>#pragma inline=forced</code> will enable inlining of the function also on the Medium optimization level.</p> |                                                                                          |
| See also    | <i>Inlining functions</i> , page 87.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                          |

## language

|             |                                                                      |                                                                                                                                                                                                                                                                                 |
|-------------|----------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma language={extended default save restore}</code>        |                                                                                                                                                                                                                                                                                 |
| 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 254 and `--strict`, page 270.

## location

Syntax

```
#pragma location={address|NAME}
```

Parameters

|                |                                                                                                      |
|----------------|------------------------------------------------------------------------------------------------------|
| <i>address</i> | The absolute address of the global or static variable for which you want an absolute location.       |
| <i>NAME</i>    | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker. |

Description

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive. Do not place variables that would normally be in different segments (for example, variables declared as `__no_init` and variables declared as `const`) in the same named segment.

**Example**

```
#pragma location=0xFFFF2000
__no_init volatile char PORT1; /* PORT1 is located at address
                                0xFFFF2000 */

#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 */
```

**See also** *Controlling data and function placement in memory*, page 214 and *Placing user-defined segments*, page 108.

## message

**Syntax** `#pragma message(message)`

**Parameters**

*message*                      The message that you want to direct to the standard output stream.

**Description**                      Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

**Example**

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

## 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 292.

**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**

```
#pragma object_attribute=__no_init
char bar;
```

is equivalent to:

```
__no_init char bar;
```

**See also** *General syntax rules for extended keywords*, page 289.

## optimize

**Syntax** `#pragma optimize=[goal] [level] [no_optimization...]`

### Parameters

|                        |                                                                                                                                                                                                                                                                                                                                      |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>goal</i>            | Choose between:<br>size, optimizes for size<br>balanced, optimizes balanced between speed and size<br>speed, optimizes for speed.<br>no_size_constraints, optimizes for speed, but relaxes the normal restrictions for code size expansion.                                                                                          |
| <i>level</i>           | Specifies the level of optimization; choose between none, low, medium, or high.                                                                                                                                                                                                                                                      |
| <i>no_optimization</i> | Disables one or several optimizations; choose between:<br>no_code_motion, disables code motion<br>no_cse, disables common subexpression elimination<br>no_inline, disables function inlining<br>no_tbaa, disables type-based alias analysis<br>no_unroll, disables loop unrolling<br>no_scheduling, disables instruction scheduling. |

**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 221.

## pack

#### Syntax

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop} [, name] [, n])
```

#### Parameters

|             |                                                                      |
|-------------|----------------------------------------------------------------------|
| <i>n</i>    | Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16      |
| Empty list  | Restores the structure alignment to default                          |
| push        | Sets a temporary structure alignment                                 |
| pop         | Restores the structure alignment from a temporarily pushed alignment |
| <i>name</i> | An optional pushed or popped alignment label                         |

#### Description

Use this pragma directive to specify the maximum alignment of `struct` and `union` members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or the end of the compilation unit.

**Note:** This can result in significantly larger and slower code when accessing members of the structure.

See also *Structure types*, page 282.

## **\_\_printf\_args**

|             |                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma __printf_args</code>                                                                                                                                                                                        |
| Description | Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct. |
| Example     | <pre>#pragma __printf_args int printf(char const *,...);  void PrintNumbers(unsigned short x) {     printf("%d", x); /* Compiler checks that x is an integer */ }</pre>                                                   |

## **public\_equ**

|             |                                                                                   |                                                                          |
|-------------|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| Syntax      | <code>#pragma public_equ="symbol", value</code>                                   |                                                                          |
| 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     | <code>#pragma public_equ="MY_SYMBOL", 0x123456</code>                             |                                                                          |
| See also    | <code>--public_equ</code> , page 268.                                             |                                                                          |

## **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 segment it resides in.</p> |
| Example     | <pre>const char copyright[] = "Copyright by me";  #pragma required=copyright int main() {     /* Do something here. */ }</pre> <p>Even if the <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           | <table> <tr> <td style="vertical-align: top;"><code>"key"</code></td> <td>A text string that specifies the runtime model attribute.</td> </tr> <tr> <td style="vertical-align: top;"><code>"value"</code></td> <td>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.</td> </tr> </table>                                                                                                                                                                                                                                                                                                                                                     | <code>"key"</code> | A text string that specifies the runtime model attribute. | <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. |
| <code>"key"</code>   | A text string that specifies the runtime model attribute.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                    |                                                           |                      |                                                                                                                                                                   |
| <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.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                    |                                                           |                      |                                                                                                                                                                   |
| Description          | <p>Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.</p> <p>This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value <code>*</code>. It can, however, be useful to state explicitly that the module can handle any runtime model.</p> <p>A module can have several runtime model definitions.</p> <p><b>Note:</b> The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.</p> |                    |                                                           |                      |                                                                                                                                                                   |
| Example              | <code>#pragma rtmodel="I2C", "ENABLED"</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                    |                                                           |                      |                                                                                                                                                                   |

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also *Checking module consistency*, page 154.

## **\_\_scanf\_args**

**Syntax** `#pragma __scanf_args`

**Description** Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct.

**Example**

```
#pragma __scanf_args
int scanf(char const *,...);

int GetNumber()
{
    int nr;
    scanf("%d", &nr); /* Compiler checks that
                       the argument is a
                       pointer to an integer */

    return nr;
}
```

## **segment**

**Syntax** `#pragma segment="NAME" [__memoryattribute] [align]`  
*alias*  
`#pragma section="NAME" [__memoryattribute] [align]`

**Parameters**

|                          |                                                                                                                              |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <i>NAME</i>              | The name of the segment.                                                                                                     |
| <i>__memoryattribute</i> | An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used. |
| <i>align</i>             | Specifies an alignment for the segment. The value must be a constant integer expression to the power of two.                 |

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Use this pragma directive to define a segment name that can be used by the segment operators <code>__segment_begin</code>, <code>__segment_end</code>, and <code>__segment_size</code>. All segment declarations for a specific segment must have the same memory type attribute and alignment.</p> <p>The <code>align</code> and the <code>__memoryattribute</code> parameters are only relevant when used together with the segment operators <code>__segment_begin</code>, <code>__segment_end</code>, and <code>__segment_size</code>. If you consider using <code>align</code> on an individual variable to achieve a higher alignment, you must instead use the <code>#pragma data_alignment</code> directive.</p> <p>If an optional memory attribute is used, the return type of the segment operators <code>__segment_begin</code> and <code>__segment_end</code> is:</p> <pre>void __memoryattribute *.</pre> <p><b>Note:</b> To place variables or functions in a specific segment, use the <code>#pragma location</code> directive or the <code>@</code> operator.</p> |
| Example     | <pre>#pragma segment="MYDATA17" __data17 4</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| See also    | <i>Dedicated segment operators</i> , page 191. For more information about segments, see the chapters <i>Linking overview</i> and <i>Linking your application</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

## shadow\_registers

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                   |                                                              |                   |                                                                              |                   |                                                                              |                   |                                                                                                                                                                                                                                                                                                    |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|--------------------------------------------------------------|-------------------|------------------------------------------------------------------------------|-------------------|------------------------------------------------------------------------------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax            | <pre>#pragma shadow_registers={none half full mask}</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                   |                                                              |                   |                                                                              |                   |                                                                              |                   |                                                                                                                                                                                                                                                                                                    |
| Parameters        | <table> <tr> <td style="vertical-align: top;"><code>none</code></td> <td>None of the registers are shadowed, except the stack pointer</td> </tr> <tr> <td style="vertical-align: top;"><code>half</code></td> <td>The registers R8–R12 and R14 are implemented as shadow registers in hardware</td> </tr> <tr> <td style="vertical-align: top;"><code>full</code></td> <td>The registers R0–R12 and R14 are implemented as shadow registers in hardware</td> </tr> <tr> <td style="vertical-align: top;"><code>mask</code></td> <td>A bitmask where each set bit represents a register that is shadowed. For example, <code>0x5F00</code> is equivalent with the parameter <code>half</code>, and <code>0x0001</code> means that only R0 is shadowed. Supplying a bit mask with bit 13 or bit 15 set will result in an error message.</td> </tr> </table> | <code>none</code> | None of the registers are shadowed, except the stack pointer | <code>half</code> | The registers R8–R12 and R14 are implemented as shadow registers in hardware | <code>full</code> | The registers R0–R12 and R14 are implemented as shadow registers in hardware | <code>mask</code> | A bitmask where each set bit represents a register that is shadowed. For example, <code>0x5F00</code> is equivalent with the parameter <code>half</code> , and <code>0x0001</code> means that only R0 is shadowed. Supplying a bit mask with bit 13 or bit 15 set will result in an error message. |
| <code>none</code> | None of the registers are shadowed, except the stack pointer                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                   |                                                              |                   |                                                                              |                   |                                                                              |                   |                                                                                                                                                                                                                                                                                                    |
| <code>half</code> | The registers R8–R12 and R14 are implemented as shadow registers in hardware                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                   |                                                              |                   |                                                                              |                   |                                                                              |                   |                                                                                                                                                                                                                                                                                                    |
| <code>full</code> | The registers R0–R12 and R14 are implemented as shadow registers in hardware                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                   |                                                              |                   |                                                                              |                   |                                                                              |                   |                                                                                                                                                                                                                                                                                                    |
| <code>mask</code> | A bitmask where each set bit represents a register that is shadowed. For example, <code>0x5F00</code> is equivalent with the parameter <code>half</code> , and <code>0x0001</code> means that only R0 is shadowed. Supplying a bit mask with bit 13 or bit 15 set will result in an error message.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                   |                                                              |                   |                                                                              |                   |                                                                              |                   |                                                                                                                                                                                                                                                                                                    |
| Description       | Use this pragma directive when declaring an interrupt function to specify whether any <i>shadow registers</i> are available in the AVR32 device that is used. This means that the                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                   |                                                              |                   |                                                                              |                   |                                                                              |                   |                                                                                                                                                                                                                                                                                                    |

interrupt routine can execute without having to save the contents of the registers that are used, because the shadow registers replace the ordinary registers.

If an interrupt function is declared without a preceding `#pragma shadow_registers` directive, the compiler will assume that no registers are shadowed, and that any used registers must be saved. This reduces the code size and improves the execution speed of the interrupt function.

**Note:** Not all shadowing modes are available for all devices and situations. For example, the none shadowing mode is not available when AT32UC3A0512 is selected, because registers are at least half-shadowed for all UC interrupt events. For this device, specifying none as an argument to the pragma directive, by default sets the shadowing mode to half.

The `pc` and `sp` registers are never shadowed.

#### Example

```
#pragma shadow_registers=half
__interrupt my_interrupt(void)
{
    ...
}
```

## STDC CX\_LIMITED\_RANGE

Syntax `#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}`

#### Parameters

|         |                                                    |
|---------|----------------------------------------------------|
| ON      | Normal complex mathematic formulas can be used.    |
| OFF     | Normal complex mathematic formulas cannot be used. |
| DEFAULT | Sets the default behavior, that is OFF.            |

#### Description

Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for \* (multiplication), / (division), and abs.

**Note:** This directive is required by Standard C. The directive is recognized but has no effect in the compiler.

## STDC FENV\_ACCESS

Syntax `#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}`

|             |                                                                                                               |                                                                                                                |
|-------------|---------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| Parameters  | ON                                                                                                            | Source code accesses the floating-point environment. Note that this argument is not supported by the compiler. |
|             | OFF                                                                                                           | Source code does not access the floating-point environment.                                                    |
|             | DEFAULT                                                                                                       | Sets the default behavior, that is OFF.                                                                        |
| Description | Use this pragma directive to specify whether your source code accesses the floating-point environment or not. |                                                                                                                |
|             | <b>Note:</b> This directive is required by Standard C.                                                        |                                                                                                                |

## STDC FP\_CONTRACT

|             |                                                                                                                                                               |                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma STDC FP_CONTRACT {ON OFF DEFAULT}</code>                                                                                                        |                                                                                                                               |
| Parameters  | ON                                                                                                                                                            | The compiler is allowed to contract floating-point expressions.                                                               |
|             | OFF                                                                                                                                                           | The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler. |
|             | DEFAULT                                                                                                                                                       | Sets the default behavior, that is ON.                                                                                        |
| Description | Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C. |                                                                                                                               |
| Example     | <code>#pragma STDC FP_CONTRACT=ON</code>                                                                                                                      |                                                                                                                               |

## type\_attribute

|             |                                                                                                                                                                                                        |  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>#pragma type_attribute=type_attr[ type_attr...]</code>                                                                                                                                           |  |
| Parameters  | For information about type attributes that can be used with this pragma directive, see <i>Type attributes</i> , page 289.                                                                              |  |
| Description | Use this pragma directive to specify IAR-specific <i>type attributes</i> , which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects. |  |

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

**Example**

In this example, an `int` object with the memory attribute `__data17` is defined:

```
#pragma type_attribute=__data17
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__data17 int x;
```

**See also**

The chapter *Extended keywords*.

## vector

**Syntax**

```
#pragma vector=vector1[, vector2, vector3, ...]
```

**Parameters**

*vectorN*                      The vector number(s) of an `acall` function.

**Description**

Use this pragma directive to specify the vector(s) of an `acall` function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

**Example**

```
#pragma vector=0x14
__acall void my_function(void);
```



# Intrinsic functions

- Summary of intrinsic functions
- Descriptions of intrinsic functions

---

## Summary of intrinsic functions

There are two types of intrinsic functions available:

- Intrinsic inline functions
- ETSI macro functions

### INTRINSIC INLINE 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:

| Intrinsic function                | Description                                                       |
|-----------------------------------|-------------------------------------------------------------------|
| <code>__bit_reverse</code>        | Inserts a bit reverse (BREV) instruction                          |
| <code>__BREAKPOINT</code>         | Inserts a breakpoint (BREAKPOINT) instruction                     |
| <code>__cache_control</code>      | Inserts a cache control (CACHE) instruction                       |
| <code>__clear_status_flag</code>  | Inserts a clear status register flag (CSRF) instruction           |
| <code>__COP</code>                | Inserts a coprocessor (COP) instruction                           |
| <code>__COP_get_register32</code> | Inserts a read coprocessor register (MVCR.w) instruction          |
| <code>__COP_get_register64</code> | Inserts a read coprocessor register (MVCR.d) instruction          |
| <code>__COP_get_registers</code>  | Inserts a read multiple coprocessor register (STCM.w) instruction |

Table 40: Intrinsic functions summary

| <b>Intrinsic function</b>           | <b>Description</b>                                                               |
|-------------------------------------|----------------------------------------------------------------------------------|
| <code>__COP_set_registers</code>    | Inserts a write coprocessor register (MVRC.w) instruction                        |
| <code>__COP_set_register32</code>   | Inserts a write coprocessor register (MVRC.d) instruction                        |
| <code>__COP_set_register64</code>   | Inserts a write multiple coprocessor register (LDCM.w) instruction               |
| <code>__count_leading_zeros</code>  | Returns the number of bits set to zero, starting from the most significant bit   |
| <code>__count_trailing_zeros</code> | Returns the number of bits set to zero, starting from the least significant bit. |
| <code>__disable_interrupt</code>    | Disables interrupts                                                              |
| <code>__enable_interrupt</code>     | Enables interrupts                                                               |
| <code>__exchange_memory</code>      | Inserts a memory exchange (XCHG) instruction                                     |
| <code>__get_debug_register</code>   | Inserts an MFDR instruction                                                      |
| <code>__get_interrupt_state</code>  | Returns the interrupt state                                                      |
| <code>__get_system_register</code>  | Inserts a read from system register (MFSR) instruction                           |
| <code>__get_user_context</code>     | Inserts a store user context (STMTS) instruction                                 |
| <code>__max</code>                  | Inserts a MAX instruction                                                        |
| <code>__min</code>                  | Inserts a MIN instruction                                                        |
| <code>__no_operation</code>         | Inserts a NOP instruction                                                        |
| <code>__prefetch_cache</code>       | Inserts a cache prefetch (PREF) instruction                                      |
| <code>__read_TLB_entry</code>       | Inserts an MMU table read (TLBR) instruction                                     |
| <code>__search_TLB_entry</code>     | Inserts an MMU table search (TLBS) instruction                                   |
| <code>__set_debug_register</code>   | Inserts an MTDR instruction                                                      |
| <code>__set_interrupt_state</code>  | Restores the interrupt state                                                     |
| <code>__set_status_flag</code>      | Insert a set status register flag (SSRF) instruction                             |
| <code>__set_system_register</code>  | Inserts a write to system register (MTSR) instruction                            |

Table 40: Intrinsic functions summary (Continued)

| Intrinsic function                      | Description                                                                    |
|-----------------------------------------|--------------------------------------------------------------------------------|
| <code>__set_user_context</code>         | Inserts a load user context ( <code>LDMTS</code> ) instruction                 |
| <code>__signed_saturate</code>          | Saturates the first parameter so that it fits in a word-size signed bitfield   |
| <code>__sleep</code>                    | Inserts a <code>SLEEP</code> instruction                                       |
| <code>__store_conditional</code>        | Inserts a conditional store ( <code>STCOND</code> ) instruction                |
| <code>__swap_bytes</code>               | Inserts a <code>SWAP.b</code> instruction                                      |
| <code>__swap_bytes_in_halfwords</code>  | Inserts a <code>SWAP.bh</code> instruction                                     |
| <code>__swap_halfwords</code>           | Inserts a <code>SWAP.h</code> instruction                                      |
| <code>__synchronize_write_buffer</code> | Inserts a synchronize write buffer ( <code>SYNC</code> ) instruction           |
| <code>__test_status_flag</code>         | Insert a get status register flag ( <code>CSRFCZ</code> ) instruction          |
| <code>__unsigned_saturate</code>        | Saturates the first parameter so that it fits in a word-size unsigned bitfield |
| <code>__write_TLB_entry</code>          | Inserts an MMU table write ( <code>TLBW</code> ) instruction                   |

Table 40: Intrinsic functions summary (Continued)

## SUMMARY AND DESCRIPTION OF ETSI FUNCTIONS

The ETSI interface is designed for telecommunications applications, but can be used with any type of application you are building. The preprocessor macros will expand to C functions or intrinsic functions that correspond to the DSP operations of the AVR32 microprocessor.

The following table lists the ETSI functions:

| ETSI function          | Syntax                                                   | Description                              |
|------------------------|----------------------------------------------------------|------------------------------------------|
| <code>abs_s</code>     | <code>q15 abs_s(q15 x);</code>                           | Saturated 16-bit absolute value of $x$   |
| <code>add</code>       | <code>q15 add(q15 x, q15 y);</code>                      | Saturated 16-bit addition                |
| <code>div_s</code>     | <code>q15 div_s(q15 x, q15 y);</code>                    | Saturated 16-bit division                |
| <code>extract_h</code> | <code>unsigned short<br/>extract_h(unsigned int);</code> | Extracts high 16-bit part of 32-bit word |
| <code>extract_l</code> | <code>unsigned short<br/>extract_l(unsigned int);</code> | Extracts low 16-bit part of 32-bit word  |

Table 41: ETSI functions summary

| ETSI function | Syntax                                       | Description                                             |
|---------------|----------------------------------------------|---------------------------------------------------------|
| L_abs         | q31 L_abs(q31 x);                            | Saturated 32-bit absolute value of x                    |
| L_add         | q31 L_add(q31 x, q31 y);                     | Saturated 32-bit addition                               |
| L_add_c       | q31 L_add_c(q31 x,<br>q31 y);                | 32-bit add with carry and overflow                      |
| L_deposit_h   | unsigned int<br>L_deposit_h(unsigned short); | Deposits value in high 16-bit part of result            |
| L_deposit_l   | unsigned int<br>L_deposit_l(unsigned short); | Deposits value in low 16-bit part of result             |
| L_mac         | q31 L_mac(q31 x, q15 y,<br>q15 z);           | Saturated multiply/accumulate                           |
| L_macNs       | q31 L_macNs(q31 x, q15 y,<br>q15 z);         | Multiply/accumulate with carry and overflow             |
| L_msu         | q31 L_msu(q31 x, q15 y,<br>q15 z);           | Saturated multiply/subtract                             |
| L_msuNs       | q31 L_msuNs(q31 x, q15 y,<br>q15 z);         | Multiply/subtract with carry and overflow               |
| L_mult        | q31 L_mult(q15 x, q15 y);                    | Saturated 16-bit multiplication with 32-bit product     |
| L_negate      | q31 L_negate(q31 x);                         | Saturated 32-bit negation                               |
| L_sat         | q31 L_sat(q31 x);                            | Saturate value depending on previous overflow and carry |
| L_shl         | q31 L_shl(q31 x, short y);                   | Saturated 32-bit left shift                             |
| L_shr         | q31 L_shr(q31 x, short y);                   | Saturated 32-bit right shift                            |
| L_shr_r       | q31 L_shr_r(q31 x,<br>short y);              | Saturated rounded 32-bit right shift                    |
| L_sub         | q31 L_sub(q31 x, q15 y);                     | Saturated 32-bit subtraction                            |
| L_sub_c       | q31 L_sub_c(q31 x,<br>q31 y);                | 32-bit subtract with carry and overflow                 |
| mac_r         | q15 mac_r(q31 x, q15 y,<br>q15 z);           | Saturated rounded multiply/accumulate                   |
| msu_r         | q15 msu_r(q31 x, q15 y,<br>q15 z);           | Saturated rounded multiply/subtract                     |
| mult          | q15 mult(q15 x, q15 y);                      | Saturated 16-bit multiplication                         |

Table 41: ETSI functions summary (Continued)

| ETSI function | Syntax                     | Description                             |
|---------------|----------------------------|-----------------------------------------|
| mult_r        | q15 mult_r(q15 x, q15 y);  | Saturated rounded 16-bit multiplication |
| negate        | q15 negate(q15 x);         | Saturated 16-bit negation               |
| norm_s        | q15 norm_s(q15 x);         | Normalizes 16-bit                       |
| norm_l        | q15 norm_l(q31 x);         | Normalizes 32-bit                       |
| round         | q15 round(q31 x);          | Rounds biased                           |
| shl           | q15 shl(q15 x, short y);   | Saturated 16-bit left shift             |
| shr           | q15 shr(q15 x, short y);   | Saturated 16-bit right shift            |
| shr_r         | q15 shr_r(q15 x, short y); | Saturated rounded 16-bit right shift    |
| sub           | q15 sub(q15 x, q15 y);     | Saturated 16-bit subtraction            |

Table 41: ETSI functions summary (Continued)

**Note:** In the table, q15 represents a 16-bit fixed-point value stored in a variable of type `int16_t` and q31 represents a 32-bit fixed-point value stored in a variable of type `int32_t`.

To use ETSI functions in an application, include the header file `etsi.h`.

## Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

### `__bit_reverse`

Syntax `unsigned int __bit_reverse(unsigned int);`

Description Inserts a bit reverse (BREV) instruction which reverses the bit pattern of the argument.

**Note:** The compiler constant-folds the parameter at optimization levels above Low.

Example 

```
unsigned int x = 0x12345678;
x = __bit_reverse(x);
/* x now has the value 0x1E6A2C48 */
```

### `__BREAKPOINT`

Syntax `void __BREAKPOINT(void);`

**Description** Inserts a `BREAKPOINT` instruction. In a hardware debug environment, the `BREAKPOINT` instruction will switch to debug mode.

In all normal cases your debug environment should set breakpoints for you, so do not expect breakpoint instructions that you place in your application code to work properly with the debug environment unless specified in the documentation.

## `__cache_control`

**Syntax** `void __cache_control(void __data32 *, unsigned int);`

**Description** Inserts a cache control (`CACHE`) instruction. The two arguments will be passed over to the `CACHE` instruction unchanged.

For details about the how the `CACHE` instruction works, see the chip manufacturer's documentation.

**Example**

```
/* Second argument specifies data cache */
__cache_control(myPointer, (1 << 3));
```

## `__clear_status_flag`

**Syntax** `void __clear_status_flag(unsigned int);`

**Description** Insert a clear status register flag (`CSRFB`) instruction. The instruction can be used for clearing a selected flag bit in the status register. Note that many bits are modified naturally during execution, for instance, the `C` and `Z` bits. If you clear these bits, you cannot rely on them to keep their values. This intrinsic function is best suited for modifying special bits like the interrupt level mask bits.

The argument is the bit number, where bit 0 is the least significant bit (the carry flag). The possible argument range is 0–31.

**Note:** The status register can also be accessed as a special function register (SFR) if you use the provided include files, for example, `ioavr32.h`.

## `__COP`

**Syntax**

```
void __COP(unsigned char coproNumber, unsigned char CPdestReg,
           unsigned char CPsrcReg1, unsigned char CPsrcReg2,
           unsigned char CPOperation);
```

**Description** Inserts a coprocessor (COP) instruction. The coprocessor instructions are highly specific to the coprocessor in question. For details, read the documentation for your coprocessor.

**Example**

```
__COP(0 /* coprocessor number */,
      3 /* destination register in coprocessor */,
      6, 7 /* source registers in coprocessor */,
      5 /* operation number in coprocessor */);
```

## \_\_COP\_get\_register32

**Syntax** `long __COP_get_register32(unsigned char coproNumber,  
unsigned char CPsrcReg);`

**Description** Inserts a read coprocessor register (MVCR.w) instruction. The contents of the specified 32-bit coprocessor register will be read and returned as a function value.

**Example**

```
/* Read register 3 of coprocessor 0 */
long x = __COP_get_register32(0, 3);
```

## \_\_COP\_get\_register64

**Syntax** `long long __COP_get_register64(unsigned char coproNumber,  
unsigned char CPsrcReg);`

**Description** Inserts a read coprocessor register (MVCR.d) instruction. The contents of the specified pair of 32-bit coprocessor registers will be read and returned as a 64-bit function value.

**Example**

```
/* Read register 4 and 5 of coprocessor 0 */
long long x = __COP_get_register64(0, 4);
```

## \_\_COP\_get\_registers

**Syntax** `void __COP_get_registers(unsigned char coproNumber,  
unsigned short CPregMask,  
long __data32 * destination);`

**Description** Inserts a write multiple coprocessor register (STCM.w) instruction. The registers specified in the mask will be read and stored in the destination buffer.

**Example**

```
/* Transfer the contents of registers 2 - 5 of coprocessor 0 to
buf */
long buf[4];
__COP_get_registers(0, 0x3C /* 00111100b */, buf);
```

## \_\_COP\_set\_registers

|             |                                                                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>void __COP_set_registers(unsigned char coproNumber,                         unsigned short CPregMask,                         long __data32 * destination);</pre> |
| Description | Inserts a write multiple coprocessor register (STCM.w) instruction. The registers specified in the mask will be read and stored in the destination buffer.             |
| Example     | <pre>/* Transfer the contents of registers 2 - 5 of coprocessor 0 to buf */ long buf[4]; __COP_get_registers(0, 0x3C /* 00111100b */, buf);</pre>                      |

## \_\_COP\_set\_register32

|             |                                                                                                                                                          |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>void __COP_set_register32(unsigned char coproNumber,                           unsigned char CPsrcReg, long value);</pre>                           |
| Description | Inserts a write coprocessor register (MVRC.w) instruction. The contents of the specified 32-bit coprocessor register will be set to the specified value. |
| Example     | <pre>/* Set a new value on register 3 of coprocessor 0 */ __COP_set_register32(0, 3, 0x01234567L);</pre>                                                 |

## \_\_COP\_set\_register64

|             |                                                                                                                                                                                                                                     |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>void __COP_set_register64(unsigned char coproNumber,                           unsigned char CPsrcReg, long long value);</pre>                                                                                                 |
| Description | Inserts a write coprocessor register (MVRC.d) instruction. The contents of the specified pair of 32-bit coprocessor registers will be set to the specified value. Remember that a long long constant must have the suffix LL or ll. |
| Example     | <pre>/* Set a new value on register 4 and 5 of coprocessor 0 */ __COP_set_register64(0, 3, 0x0123456789ABCDEFLL);</pre>                                                                                                             |

## \_\_count\_leading\_zeros

|        |                                                              |
|--------|--------------------------------------------------------------|
| Syntax | <pre>unsigned int __count_leading_zeros(unsigned int);</pre> |
|--------|--------------------------------------------------------------|



**Description** Returns the number of bits set to zero, starting from the most significant bit until it reaches a 1.

**Note:** The compiler constant-folds the parameter at optimization levels above Low.

**Example**

```
__count_leading_zeros(0xFFFFFFFF)=0
__count_leading_zeros(0x08000000)=4
__count_leading_zeros(0x00000000)=32
```

## \_\_count\_trailing\_zeros

**Syntax** `unsigned int __count_trailing_zeros(unsigned int);`

**Description** Returns the number of bits set to zero, starting from the least significant bit until it reaches a 1.

**Note:** The compiler constant-folds the parameter at optimization levels above Low.

**Example**

```
__count_trailing_zeros(0xFFFFFFFF)=0
__count_trailing_zeros(0x08000000)=27
__count_trailing_zeros(0x00000000)=32
```

## \_\_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.

## \_\_exchange\_memory

**Syntax** `unsigned int __exchange_memory(unsigned int __data32 *, unsigned int);`

**Description** Inserts a memory exchange (`XCHG`) instruction. The value of the second parameter is stored at the specified address, and the function returns the previous value of the word

specified by the address. Because this instruction will both read and update a variable in memory in one atomic operation, it can be used to implement a semaphore.

#### Example

```
enum semaphoreValue = {busy, free};
extern unsigned int mySemaphore;
semaphoreValue temp;
temp = (semaphoreValue) __exchange_memory(&mySemaphore, busy);
if (temp == free)
{
    /* Semaphore was free, we can do the operation
       that was guarded by the semaphore */
}
```

### **\_\_get\_debug\_register**

Syntax `unsigned int __get_debug_register(unsigned short);`

Description Inserts an MFDR instruction.

### **\_\_get\_interrupt\_state**

Syntax `__istate_t __get_interrupt_state(void);`

Description Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

#### Example

```
#include "intrinsics.h"

void CriticalFn()
{
    __istate_t s = __get_interrupt_state();
    __disable_interrupt();

    /* Do something here. */

    __set_interrupt_state(s);
}
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

## \_\_get\_system\_register

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>unsigned int __get_system_register(unsigned short);</pre>                                                                                                                                                                                                                                                                                                                                                                           |
| Description | <p>Inserts a read from system register (MFSR) instruction. The argument is the system register address (0, 4, 8, ... up to 1020), not the register number (0, 1, 2, ...), in exactly the same way as the argument to the assembler instruction MFSR.</p> <p><b>Note:</b> Many system registers can also be accessed as a special function register (SFR) if you use the provided include files, for example, <code>ioavr32.h</code>.</p> |
| Example     | <pre>/* EVBA has number 8, address 32 */ unsigned int evba = __get_system_register(32);</pre>                                                                                                                                                                                                                                                                                                                                            |

## \_\_get\_user\_context

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>void __get_user_context(unsigned short,                         unsigned long __data32 *);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Description | <p>Inserts a store user context (STMTS) instruction. The STMTS instruction will transfer the contents of the specified application context registers to the specified buffer. It is typically used within an operating system when performing a task switch to save the values of the application mode registers of the previously executing task.</p> <p><b>Note:</b> You should call this intrinsic function within an environment where banked registers are used, to avoid the currently executing C code to modify the application context registers.</p> |
| Example     | <pre>unsigned long registerFile[16]; __get_user_context(0xFFFF, registerFile);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

## \_\_max

|             |                                                                                                                                   |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>signed int __max(signed int, signed int);</pre>                                                                              |
| Description | <p>Inserts a MAX instruction.</p> <p><b>Note:</b> The compiler constant-folds the parameter at optimization levels above Low.</p> |

## \_\_min

|        |                                                      |
|--------|------------------------------------------------------|
| Syntax | <pre>signed int __min(signed int, signed int);</pre> |
|--------|------------------------------------------------------|

Description Inserts a `MIN` instruction.  
**Note:** The compiler constant-folds the parameter at optimization levels above Low.

## `__no_operation`

Syntax `void __no_operation(void);`

Description Inserts a `NOP` instruction.

## `__prefetch_cache`

Syntax `void __prefetch_cache(void __data32 *);`

Description Inserts a cache prefetch (`PREF`) instruction. The `PREF` instruction can be used to direct the cache, if present, to prefetch information from memory. For details, see the chip manufacturer's documentation.

Example 

```
char buffer[8192];
__prefetch_cache(buffer);
```

## `__read_TLB_entry`

Syntax `void __read_TLB_entry(void);`

Description Inserts an MMU table read (`TLBR`) instruction. For details about how to use the MMU, see the chip manufacturer's documentation.

Example 

```
/* Use SFR declarations from ioavr32.h */
#include <ioavr32.h>
/* Read from instruction TLB */
TLBEHI_bit.I = 1;
/* We want to read entry 30 from the instruction TLB */
MMUCR_bit.IRP = 30;
/* Read the TLB entry */
__read_TLB_entry();
/* The entry has now been transferred to TLBEHI and TLBELO */
```

## `__search_TLB_entry`

Syntax `void __search_TLB_entry(void);`

**Description** Inserts an MMU table search (TLBS) instruction. The entry to search for is specified by first writing to the MMU interface registers TLBEHI and TLBELO.

For details about how to use the MMU, see the chip manufacturer's documentation.

**Example**

```

/* Use SFR declarations from ioavr32.h */
#include <ioavr32.h>
/* Set up the values to search for */
TLBEHI = ...
TLBELO = ...
/* Search instruction TLB */
TLBEHI_bit.I = 1;
/* Search for a matching TLB entry */
__search_TLB_entry();
/* Check if desired entry found */
if (MMUCR_bit.N == 0)
{
    /* The entry was found */
}

```

## \_\_set\_debug\_register

**Syntax** `void __set_debug_register(unsigned short, unsigned int);`

**Description** Inserts an MTDR instruction.

## \_\_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 338.

## \_\_set\_status\_flag

**Syntax** `void __set_status_flag(unsigned int);`

**Description** Inserts a set status register flag (SSRF) instruction. The instruction can be used to set a selected flag bit in the status register. Note that many bits are modified naturally during execution, for instance, the C and Z bits. If you set these bits, you cannot rely on them to

keep the value. This intrinsic function is best suited for modifying special bits like the interrupt level mask bits.

The argument is the bit number, where bit 0 is the least significant bit (the carry flag). The possible argument range is 0–31.

**Note:** The status register can also be accessed as a special function register (SFR) if you use the provided include files, for example, `ioavr32.h`.

Example

```
/* Set interrupt level 0 mask */
__set_status_flag(17);
```

## **\_\_set\_system\_register**

Syntax `void __set_status_flag(unsigned short, unsigned int);`

Description Inserts a write to system register (MCSR) instruction. The first argument is the system register address (0, 4, 8, ... up to 1020), not the register number (0, 1, 2, ...), in exactly the same way as the argument to the assembler instruction `MCSR`. The second argument is the new value to write to the system register.

**Note:** Many system registers can also be accessed as a special function register (SFR) if you use the provided include files, for example, `ioavr32.h`.

Example

```
/* EVBA has number 8, address 32 */
unsigned int evba = 0xFFFFF00;
__set_system_register(32, evba);
```

## **\_\_set\_user\_context**

Syntax `void __set_user_context(unsigned short, unsigned long __data32 *);`

Description Inserts a load user context (LDMTS) instruction. The `LDMTS` instruction will transfer the contents of the specified buffer to the specified application context registers. It is typically used within an operating system when performing a task switch to restore the previous values of the application mode registers of the task to switch to.

**Note:** You should call this intrinsic function within an environment where banked registers are used, to avoid the currently executing C code to modify the restored registers.

Example

```
unsigned long registerFile[16];
__set_user_context(0xFFFF, registerFile);
```

## \_\_signed\_saturate

|             |                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>signed int __signed_saturate(signed int x, unsigned char w);</code>                                                                                                                                                                                                                                                                                           |
| Description | Saturates the first parameter so that it fits in a word-size signed bitfield. That is, the range of the result will be $-2^{(N-1)} \dots 2^{(N-1)}-1$ .<br><br>The compiler tries to generate either a <code>SATS</code> instruction or a <code>SATRND</code> s instruction. It also tries to incorporate an arithmetic right shift into the generated instruction. |
| Example     | <code>__signed_saturate((x + (1 &lt;&lt; 2)) &gt;&gt; 3, 16);</code><br><br>This should generate (assuming that a copy of <code>x</code> is in <code>R0</code> ):<br><code>SATRND R0 &gt;&gt; 3, 16</code>                                                                                                                                                          |

## \_\_sleep

|             |                                           |
|-------------|-------------------------------------------|
| Syntax      | <code>void __sleep(void);</code>          |
| Description | Inserts a <code>SLEEP</code> instruction. |

## \_\_store\_conditional

|             |                                                                                                                                                                                                                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>bool __store_conditional(unsigned int * addr, unsigned int val);</code>                                                                                                                                                                                                                                                  |
| Description | Inserts a <code>STCOND</code> instruction and returns true if the store was performed. This function can be used for implementing atomic operations.<br><br>To work, the application must first have set the <code>L</code> flag in the status register to 1 by calling the <code>__set_status_flag</code> intrinsic function. |

**Example**

```

unsigned int timer;
bool flag = false;
extern volatile unsigned int my_volatile_timer;
do
{
    __set_status_flag(5 /* The L flag */);
    timer = my_volatile_counter;
    flag = false;
    if (--timer == 0)
    {
        flag = true;
        timer = TIMER_VALUE;
    }
    // Conditionally write back the new timer value
    // unless an interrupt or exception has occurred,
    // in which case we redo the code above.
} while(!__store_conditional(&my_volatile_timer, timer));
    
```

## **\_\_swap\_bytes**

**Syntax**

```
unsigned int __swap_bytes(unsigned int);
```

**Description**

Inserts a `SWAP.b` instruction.

**Note:** The compiler constant-folds the parameter at optimization levels above Low.

## **\_\_swap\_bytes\_in\_halfwords**

**Syntax**

```
unsigned int __swap_bytes_in_halfwords(unsigned int);
```

**Description**

Inserts a `SWAP.bh` instruction.

**Note:** The compiler constant-folds the parameter at optimization levels above Low.

## **\_\_swap\_halfwords**

**Syntax**

```
unsigned int __swap_halfwords(unsigned int);
```

**Description**

Inserts a `SWAP.h` instruction.

**Note:** The compiler constant-folds the parameter at optimization levels above Low.



## `__synchronize_write_buffer`

|             |                                                                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __synchronize_write_buffer(unsigned char);</code>                                                                                                         |
| Description | Inserts a synchronize write buffer ( <code>SYNC</code> ) instruction. The argument is implementation-defined. See the chip manufacturer's documentation for details. |
| Example     | <code>__synchronize_write_buffer(3);</code>                                                                                                                          |

## `__test_status_flag`

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int __test_status_flag(unsigned int);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Description | <p>Insert a get status register flag (<code>CSRFCZ</code>) instruction. The instruction can be used to test the value of a selected flag bit in the status register. Note that many bits are modified naturally during execution, for instance, the <code>C</code> and <code>Z</code> bits. Even if you test the value of these bits, you cannot draw any meaningful conclusion from the result. This intrinsic function is best suited for testing special bits like the interrupt level mask bits.</p> <p>The argument is the bit number, where bit 0 is the least significant bit (the carry flag). The possible argument range is 0–31.</p> <p><b>Note:</b> The status register can also be accessed as a special function register (SFR) if you use the provided include files, for example, <code>ioavr32.h</code>.</p> |
| Example     | <pre>/* Check interrupt level 0 mask */ int i0m = __test_status_flag(17);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

## `__unsigned_saturate`

|             |                                                                                                                                                                                                                                                                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned int __unsigned_saturate(signed int x, unsigned char w);</code>                                                                                                                                                                                                                                                                                                  |
| Description | <p>Saturates the first parameter so that it fits in a word-size unsigned bitfield. That is, the range of the result will be <math>0 \dots 2^{(N-1)} - 1</math>.</p> <p>The compiler tries to generate either a <code>SATU</code> instruction or a <code>SATRNDU</code> instruction. It also tries to incorporate an arithmetic right shift into the generated instruction.</p> |
| Example     | <pre>__unsigned_saturate((x &gt;&gt; 5), 16);</pre> <p>This should generate (assuming that a copy of <code>x</code> is in <code>R0</code>):</p> <pre>SATU R0 &gt;&gt; 5, 16</pre>                                                                                                                                                                                              |

## **\_\_write\_TLB\_entry**

Syntax

```
void __write_TLB_entry(void);
```

Description

Inserts an MMU table write (TLBW) instruction. The value to write to the TLB is specified by first writing to the MMU interface registers TLBEHI and TLBELO.

For details about how to use the MMU, see the chip manufacturer's documentation.

Example

```
/* Use SFR declarations from ioavr32.h */
#include <ioavr32.h>
/* Set up the values to enter into the TLB */
TLBEHI = ...
TLBELO = ...
/* Write to the instruction TLB */
TLBEHI_bit.I = 1;
/* Write the new entry to the TLB */
__write_TLB_entry();
```

# 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 AVR32 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 348.
- 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 248.
- 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 353.
- Preprocessor output  
Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 268.

To specify a path for an include file, use forward slashes:

```
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile", "rt");
```

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.

---

## Description of predefined preprocessor symbols

This section lists and describes the preprocessor symbols.

### **\_\_BASE\_FILE\_\_**

**Description** A string that identifies the name of the base source file (that is, not the header file), being compiled.

**See also** `__FILE__`, page 350, and `--no_path_in_file_macros`, page 262.

### **\_\_BUILD\_NUMBER\_\_**

**Description** A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.

### **\_\_CODE\_MODEL\_\_**

**Description** An integer that identifies the code model in use. The value reflects the setting of the `--code_model` option and is defined to `__CODE_MODEL_SMALL__`, `__CODE_MODEL_MEDIUM__`, or `__CODE_MODEL_LARGE__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.

### **\_\_CORE\_\_**

**Description** An integer that identifies the chip core in use. The value reflects the setting of the `--core` option and is defined to `__AVR32A__` or `__AVR32B__`. These symbolic names can be used when testing the `__CORE__` symbol.

### **\_\_CORE\_REVISION\_\_**

**Description** An integer that corresponds to the core revision of the selected CPU or core. The symbol reflects the `--core_revision` option.

### **\_\_COUNTER\_\_**

**Description** A macro that expands to a new integer each time it is expanded, starting at zero (0) and counting up.

**\_\_cplusplus**

Description An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

**\_\_DATA\_MODEL\_\_**

Description An integer that identifies the data model in use. The value reflects the setting of the `--data_model` option and is defined to `__MODEL_SMALL__` or `__MODEL_LARGE__`.

**\_\_DATE\_\_**

Description A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Oct 30 2014"

This symbol is required by Standard C.

**\_\_DEFAULT\_CODE\_SEGMENT\_\_**

Description Defined to be the default code memory type attribute, depending on the code model.

**\_\_DEFAULT\_CONST\_SEGMENT\_\_**

Description Defined to be the default data memory type attribute used for constant objects, depending on the code model.

**\_\_DEFAULT\_DATA\_SEGMENT\_\_**

Description Defined to be the default data memory type attribute used for non-constant objects, depending on the data model.

**\_\_embedded\_cplusplus**

Description An integer which is defined to 1 when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with `#ifdef` to detect

whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

## **\_\_FILE\_\_**

**Description** A string that identifies the name of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

**See also** `__BASE_FILE__`, page 348, and `--no_path_in_file_macros`, page 262.

## **\_\_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 254 and `__PRETTY_FUNCTION__`, page 352.

## **\_\_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 254 and `__PRETTY_FUNCTION__`, page 352.

## **\_\_HAS\_DSP\_INSTRUCTIONS\_\_**

**Description** An integer that is defined to 1 if DSP instructions are enabled. The symbol reflects the `--core` option, `--cpu` option, or the `--avr32_dsp_instructions=enabled` option.

## **\_\_HAS\_FPU\_INSTRUCTIONS\_\_**

**Description** An integer that is defined to a non-zero value if FPU instructions are enabled. The symbol reflects the `--avr32_fpu_instructions=enabled` option.

**\_\_HAS\_RMW\_INSTRUCTIONS\_\_**

Description An integer that is defined to 1 if RMW instructions are enabled. The symbol reflects the `--core` option, `--cpu` option, or the `--avr32_rmw_instructions=enabled` option.

**\_\_HAS\_SIMD\_INSTRUCTIONS\_\_**

Description An integer that is defined to 1 if SIMD instructions are enabled. The symbol reflects the `--core` option, `--cpu` option, or the `--avr32_simd_instructions=enabled` option.

**\_\_IAR\_SYSTEMS\_ICC\_\_**

Description An integer that identifies the IAR compiler platform. The current value is . Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.

**\_\_ICCAVR32\_\_**

Description An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for AVR32.

**\_\_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.

**\_\_PART\_\_**

Description An integer that identifies the AVR32 part in use. The symbol reflects the `--cpu` option and is defined to a symbolic name `__PART-NAME__`, reflecting the part. These symbolic names can be used when testing the `__PART__` symbol. For a list of supported parts, see the `supported_devices.html` file available in the `doc` directory.

**\_\_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 254 and `__func__`, page 350.

**\_\_STDC\_\_**

Description An integer that is set to 1, which means the compiler adheres to Standard C. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to Standard C.\* This symbol is required by Standard C.

**\_\_STDC\_VERSION\_\_**

Description An integer that identifies the version of the C standard in use. The symbol expands to 199901L, unless the `--c89` compiler option is used in which case the symbol expands to 199409L. This symbol does not apply in EC++ mode.

This symbol is required by Standard C.

**\_\_SUBVERSION\_\_**

Description An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.

**\_\_TIME\_\_**

Description A string that identifies the time of compilation in the form `"hh:mm:ss"`.

This symbol is required by Standard C.

**\_\_TIMESTAMP\_\_**

Description A string constant that identifies the date and time of the last modification of the current source file. The format of the string is the same as that used by the `asctime` standard function (in other words, `"Tue Sep 16 13:03:52 2014"`).



**\_\_VER\_\_**

## Description

An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: (100 \* the major version number + the minor version number). For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

---

## Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and Standard C directives.

**NDEBUG**

## Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

## See also

*Assert*, page 148.

**#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.



# Library functions

- Library overview
- IAR DLIB Library

For detailed reference information about the library functions, see the online help system.

---

## Library overview

**The IAR DLIB Library** is a complete library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, 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* in this guide.

### HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

### LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic project configuration*, page 54 . The linker will

include only those routines that are required—directly or indirectly—by your application.

### ALTERNATIVE MORE ACCURATE LIBRARY FUNCTIONS

The default implementation of `cos`, `sin`, `tan`, and `pow` is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named `__iar_xxx_accuratef` for float variants of the functions and `__iar_xxx_accuratel` for long double variants of the functions, and where `xxx` is `cos`, `sin`, etc.

To use any of these more accurate versions, use the `-e` linker option.

### REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcstomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `scanf`, `sscanf`, `getchar`, and `putchar`. In addition, if you are using the options `--enable_multibyte` and `--dlib_config=Full`, the `printf` and `sprintf` functions (or any variants) can also use the heap.

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.

---

## IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior for Standard C* in this guide.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code, see the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of AVR32 features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, see *Added C functionality*, page 361.

### C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Using C*.

This table lists the C header files:

| 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            |

Table 42: Traditional Standard C header files—DLIB

| Header file             | Usage                                                              |
|-------------------------|--------------------------------------------------------------------|
| <code>inttypes.h</code> | Defining formatters for all types defined in <code>stdint.h</code> |
| <code>iso646.h</code>   | Using Amendment 1— <code>iso646.h</code> standard header           |
| <code>limits.h</code>   | Testing integer type properties                                    |
| <code>locale.h</code>   | Adapting to different cultural conventions                         |
| <code>math.h</code>     | Computing common mathematical functions                            |
| <code>setjmp.h</code>   | Executing non-local <code>goto</code> statements                   |
| <code>signal.h</code>   | Controlling various exceptional conditions                         |
| <code>stdarg.h</code>   | Accessing a varying number of arguments                            |
| <code>stdbool.h</code>  | Adds support for the <code>bool</code> data type in C.             |
| <code>stddef.h</code>   | Defining several useful types and macros                           |
| <code>stdint.h</code>   | Providing integer characteristics                                  |
| <code>stdio.h</code>    | Performing input and output                                        |
| <code>stdlib.h</code>   | Performing a variety of operations                                 |
| <code>string.h</code>   | Manipulating several kinds of strings                              |
| <code>tgmath.h</code>   | Type-generic mathematical functions                                |
| <code>time.h</code>     | Converting between various time and date formats                   |
| <code>uchar.h</code>    | Unicode functionality (IAR extension to Standard C)                |
| <code>wchar.h</code>    | Support for wide characters                                        |
| <code>wctype.h</code>   | Classifying wide characters                                        |

*Table 42: Traditional Standard C header files—DLIB (Continued)*

## C++ HEADER FILES

This section lists the C++ header files:

- The C++ library header files  
The header files that constitute the Embedded C++ library.
- The C++ standard template library (STL) header files  
The header files that constitute STL for the Extended Embedded C++ library.
- The C++ C header files  
The C++ header files that provide the resources from the C library.

## The C++ library header files

This table lists the header files that can be used in Embedded C++:

| Header file               | Usage                                                                             |
|---------------------------|-----------------------------------------------------------------------------------|
| <code>complex</code>      | Defining a class that supports complex arithmetic                                 |
| <code>fstream</code>      | Defining several I/O stream classes that manipulate external files                |
| <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>new</code>          | Declaring several functions that allocate and free storage                        |
| <code>ostream</code>      | Defining the class that performs insertions                                       |
| <code>sstream</code>      | Defining several I/O stream classes that manipulate string containers             |
| <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 |

Table 43: C++ header files

## The C++ standard template library (STL) header files

The following table lists the standard template library (STL) header files that can be used in Extended Embedded C++:

| Header file             | Description                                            |
|-------------------------|--------------------------------------------------------|
| <code>algorithm</code>  | Defines several common operations on sequences         |
| <code>deque</code>      | A deque sequence container                             |
| <code>functional</code> | Defines several function objects                       |
| <code>hash_map</code>   | A map associative container, based on a hash algorithm |
| <code>hash_set</code>   | A set associative container, based on a hash algorithm |
| <code>iterator</code>   | Defines common iterators, and operations on iterators  |
| <code>list</code>       | A doubly-linked list sequence container                |
| <code>map</code>        | A map associative container                            |
| <code>memory</code>     | Defines facilities for managing memory                 |
| <code>numeric</code>    | Performs generalized numeric operations on sequences   |

Table 44: Standard template library header files

| Header file          | Description                             |
|----------------------|-----------------------------------------|
| <code>queue</code>   | A queue sequence container              |
| <code>set</code>     | A set associative container             |
| <code>slist</code>   | A singly-linked list sequence container |
| <code>stack</code>   | A stack sequence container              |
| <code>utility</code> | Defines several utility components      |
| <code>vector</code>  | A vector sequence container             |

Table 44: Standard template library 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`.

This table shows the new header files:

| Header file            | Usage                                                              |
|------------------------|--------------------------------------------------------------------|
| <code>cassert</code>   | Enforcing assertions when functions execute                        |
| <code>cctype</code>    | Classifying characters                                             |
| <code>cerrno</code>    | Testing error codes reported by library functions                  |
| <code>cfloat</code>    | Testing floating-point type properties                             |
| <code>cinttypes</code> | Defining formatters for all types defined in <code>stdint.h</code> |
| <code>climits</code>   | Testing integer type properties                                    |
| <code>locale</code>    | Adapting to different cultural conventions                         |
| <code>cmath</code>     | Computing common mathematical functions                            |
| <code>csetjmp</code>   | Executing non-local goto statements                                |
| <code>csignal</code>   | Controlling various exceptional conditions                         |
| <code>cstdarg</code>   | Accessing a varying number of arguments                            |
| <code>cstdbool</code>  | Adds support for the <code>bool</code> data type in C.             |
| <code>cstddef</code>   | Defining several useful types and macros                           |
| <code>stdint</code>    | Providing integer characteristics                                  |
| <code>stdio</code>     | Performing input and output                                        |
| <code>stdlib</code>    | Performing a variety of operations                                 |
| <code>cstring</code>   | Manipulating several kinds of strings                              |
| <code>ctime</code>     | Converting between various time and date formats                   |

Table 45: New Standard C header files—DLIB



| Header file         | Usage                       |
|---------------------|-----------------------------|
| <code>wchar</code>  | Support for wide characters |
| <code>wctype</code> | Classifying wide characters |

Table 45: New Standard C header files—DLIB (Continued)

## LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

## ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality.

The following include files provide these features:

- `fcntl.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

### **fcntl.h**

In `fcntl.h`, trap handling support for floating-point numbers is defined with the functions `fegetrapenable` and `fegettrapdisable`.

### **stdio.h**

These functions provide additional I/O functionality:

|                            |                                                                      |
|----------------------------|----------------------------------------------------------------------|
| <code>fdopen</code>        | Opens a file based on a low-level file descriptor.                   |
| <code>fileno</code>        | Gets the low-level file descriptor from the file descriptor (FILE*). |
| <code>__gets</code>        | Corresponds to <code>fgets</code> on <code>stdin</code> .            |
| <code>getw</code>          | Gets a <code>wchar_t</code> character from <code>stdin</code> .      |
| <code>putw</code>          | Puts a <code>wchar_t</code> character to <code>stdout</code> .       |
| <code>__ungetchar</code>   | Corresponds to <code>ungetc</code> on <code>stdout</code> .          |
| <code>__write_array</code> | Corresponds to <code>fwrite</code> on <code>stdout</code> .          |

## string.h

These are the additional functions defined in `string.h`:

|                         |                                                |
|-------------------------|------------------------------------------------|
| <code>strdup</code>     | Duplicates a string on the heap.               |
| <code>strcasemp</code>  | Compares strings case-insensitive.             |
| <code>strncasemp</code> | Compares strings case-insensitive and bounded. |
| <code>strlen</code>     | Bounded string length.                         |

## time.h

There are two interfaces for using `time_t` and the associated functions `time`, `ctime`, `difftime`, `gmtime`, `localtime`, and `mktime`:

- The 32-bit interface supports years from 1900 up to 2035 and uses a 32-bit integer for `time_t`. The type and function have names like `__time32_t`, `__time32`, etc. This variant is mainly available for backwards compatibility.
- The 64-bit interface supports years from -9999 up to 9999 and uses a signed `long long` for `time_t`. The type and function have names like `__time64_t`, `__time64`, etc.

In both interfaces, `time_t` starts at the year 1970.

The interfaces are defined in the system header file `time.h`.

An application can use either interface, and even mix them by explicitly using the 32- or 64-bit variants. By default, the library and the header redirect `time_t`, `time` etc. to the 32-bit variants. However, to explicitly redirect them to their 64-bit variants, define `_DLIB_TIME_USES_64` in front of the inclusion of `time.h` or `ctime`.

See also, *Time*, page 145.

`clock_t` is represented by a 32-bit integer type.

## SYMBOLS USED INTERNALLY BY THE LIBRARY

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__code`, `__data`

These symbols are used as memory attributes internally by the compiler, and they might have to be used as arguments in certain templates.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor`, `__has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

**Note:** The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.



# Segment reference

- Summary of segments
- Descriptions of segments

For more information about placement of segments, see the chapter *Linking your application*.

---

## Summary of segments

The compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

The table below lists the segments that are available in the compiler:

| Segment   | Description                                                                                           |
|-----------|-------------------------------------------------------------------------------------------------------|
| ACTAB     | Holds addresses of acall functions.                                                                   |
| CHECKSUM  | Holds the checksum generated by the linker.                                                           |
| CODE21    | Holds user application code declared <code>__code21</code> .                                          |
| CODE32    | Holds user application code declared <code>__code32</code> .                                          |
| CSTACK    | Holds the data and return address stack in application mode.                                          |
| DATA17_AC | Holds <code>__data17</code> located constant data.                                                    |
| DATA17_AN | Holds <code>__data17</code> located uninitialized data.                                               |
| DATA17_C  | Holds <code>__data17</code> constant data.                                                            |
| DATA17_I  | Holds <code>__data17</code> static and global initialized variables.                                  |
| DATA17_ID | Holds initial values for <code>__data17</code> static and global variables in <code>DATA17_I</code> . |
| DATA17_N  | Holds <code>__no_init __data17</code> static and global variables.                                    |
| DATA17_Z  | Holds zero-initialized <code>__data17</code> static and global variables.                             |
| DATA21_AC | Holds <code>__data21</code> located constant data.                                                    |
| DATA21_AN | Holds <code>__data21</code> located uninitialized data.                                               |
| DATA21_C  | Holds <code>__data21</code> constant data.                                                            |
| DATA21_I  | Holds <code>__data21</code> static and global initialized variables.                                  |

Table 46: Segment summary

| Segment      | Description                                                                                                                                |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| DATA21_ID    | Holds initial values for <code>__data21</code> static and global variables in <code>DATA21_I</code> .                                      |
| DATA21_N     | Holds <code>__no_init __data21</code> static and global variables.                                                                         |
| DATA21_Z     | Holds zero-initialized <code>__data21</code> static and global variables.                                                                  |
| DATA32_AC    | Holds <code>__data32</code> located constant data.                                                                                         |
| DATA32_AN    | Holds <code>__data32</code> located uninitialized data.                                                                                    |
| DATA32_C     | Holds <code>__data32</code> constant data.                                                                                                 |
| DATA32_I     | Holds <code>__data32</code> static and global initialized variables.                                                                       |
| DATA32_ID    | Holds initial values for <code>__data32</code> static and global variables in <code>DATA32_I</code> .                                      |
| DATA32_N     | Holds <code>__no_init __data32</code> static and global variables.                                                                         |
| DATA32_Z     | Holds zero-initialized <code>__data32</code> static and global variables.                                                                  |
| DBGREG_AC    | Holds <code>__dbgreg</code> located constant data.                                                                                         |
| DBGREG_AN    | Holds <code>__dbgreg</code> located uninitialized data.                                                                                    |
| DIFUNCT      | Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before <code>main</code> is called. |
| EVBYTES1     | Holds the code bytes of the <code>EVTA</code> segment when linking for segment translation systems.                                        |
| EVBYTES2     | Holds the code bytes of the <code>EV100</code> segment when linking for segment translation systems.                                       |
| EVBYTES3     | Holds the code bytes of the <code>EVSEG</code> segment when linking for segment translation systems.                                       |
| EVSEG        | Holds code for exception handlers.                                                                                                         |
| EVTAB        | Holds code for exception handlers.                                                                                                         |
| EV100        | Holds code for exception handlers.                                                                                                         |
| FVVEC        | Holds secure state exception vectors for FlashVault-enabled applications.                                                                  |
| HEAP         | Holds the heap used for dynamically allocated data.                                                                                        |
| HTAB         | Holds interrupt handler initialization data.                                                                                               |
| INITTAB      | Holds the segment initializer table.                                                                                                       |
| RAMCODE21    | Holds user application code declared <code>__ramfunc</code> .                                                                              |
| RAMCODE21_ID | Holds code for <code>__ramfunc</code> declared functions in <code>RAMCODE21</code> .                                                       |
| RAMCODE32    | Holds user application code declared <code>__ramfunc</code> .                                                                              |
| RAMCODE32_ID | Holds code for <code>__ramfunc</code> declared functions in <code>RAMCODE32</code> .                                                       |

Table 46: Segment summary (Continued)

| Segment     | Description                                                                             |
|-------------|-----------------------------------------------------------------------------------------|
| RESET       | Holds code executed when the processor has been reset.                                  |
| RESETCODE   | Holds the code bytes of the RESET segment when linking for segment translation systems. |
| SSTACK      | Holds the internal data and return stack in supervisor mode.                            |
| SWITCH      | Holds switch tables generated by switch statements.                                     |
| SYSREG_AC   | Holds <code>__sysreg</code> located constant data.                                      |
| SYSREG_AN   | Holds <code>__sysreg</code> located uninitialized data.                                 |
| TRACEBUFFER | Holds trace data when using NanoTrace.                                                  |

Table 46: Segment summary (Continued)

## Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives `-z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous. For information about these directives, see *Using the -Z command for sequential placement*, page 107 and *Using the -P command for packed placement*, page 107, respectively.

For each segment, the segment memory type is specified, which indicates in which type of memory the segment should be placed; see *Segment memory type*, page 90.

For information about how to define segments in the linker configuration file, see *Linking your application*, page 105.

For more information about the extended keywords mentioned here, see the chapter *Extended keywords*.

## ACTAB

### Description

Holds addresses to functions declared `__acall`.

Functions declared with the `__acall` keyword may be called with the short instruction `ACALL`, but because the same calling convention is used as for normal functions, they can also be called like normal functions.

The ACTAB segment contains a list of function addresses for functions declared with the `__acall` keyword.

The compiler automatically creates this table when a function declared with the keyword `__acall` is encountered.

|                     |                                                                                                      |
|---------------------|------------------------------------------------------------------------------------------------------|
| Segment memory type | CONST                                                                                                |
| Memory placement    | This segment can be placed anywhere in memory, but check for device-specific alignment requirements. |
| Access type         | Read-only                                                                                            |

## CHECKSUM

|                     |                                                                                                                                                                                                     |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the checksum bytes generated by the linker. This segment also holds the <code>__checksum</code> symbol. Note that the size of this segment is affected by the linker option <code>-J</code> . |
| Segment memory type | CONST                                                                                                                                                                                               |
| Memory placement    | This segment can be placed anywhere in ROM memory.                                                                                                                                                  |
| Access type         | Read-only                                                                                                                                                                                           |

## CODE21

|                     |                                                     |
|---------------------|-----------------------------------------------------|
| Description         | Holds program code declared <code>__code21</code> . |
| Segment memory type | CODE                                                |
| Memory placement    | 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFF               |
| Access type         | Read-only                                           |

## CODE32

|                     |                                                     |
|---------------------|-----------------------------------------------------|
| Description         | Holds program code declared <code>__code32</code> . |
| Segment memory type | CODE                                                |
| Memory placement    | This segment can be placed anywhere in memory.      |
| Access type         | Read-only                                           |



## CSTACK

|                     |                                                |
|---------------------|------------------------------------------------|
| Description         | Holds the internal data stack.                 |
| Segment memory type | DATA                                           |
| Memory placement    | This segment can be placed anywhere in memory. |
| Access type         | Read-write                                     |
| See also            | <i>Setting up stack memory</i> , page 109.     |

## DATA17\_AC

|             |                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__data17</code> located constant data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA17\_AN

|             |                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__no_init __data17</code> located data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA17\_C

|                     |                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------|
| Description         | Holds <code>__data17</code> constant data. This can include constant variables, string and aggregate literals, etc. |
| Segment memory type | CONST                                                                                                               |
| Memory placement    | 0x0-0x1FFFF and 0xFFFE0000-0xFFFFFFFF                                                                               |
| Access type         | Read-only                                                                                                           |

## DATA17\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__data17</code> static and global initialized variables initialized by copying from the segment <code>DATA17_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Memory placement    | 0x0-0x1FFFF and 0xFFFE0000-0xFFFFFFFF                                                                                                                                                                                                                                                                                                                                                                                                      |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## DATA17\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__data17</code> static and global variables in the <code>DATA17_I</code> segment. These values are copied from <code>DATA17_ID</code> to <code>DATA17_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | 0x0-0x1FFFF and 0xFFFE0000-0xFFFFFFFF                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

## DATA17\_N

|                     |                                                                    |
|---------------------|--------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __data17</code> variables. |
| Segment memory type | DATA                                                               |
| Memory placement    | 0x0-0x1FFFF and 0xFFFE0000-0xFFFFFFFF                              |
| Access type         | Read-write                                                         |

## DATA17\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds zero-initialized <code>__data17</code> static and global variables. The contents of this segment is declared by the system startup code.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Memory placement    | 0x0-0x1FFFFF and 0xFFFFE0000-0xFFFFFFFF                                                                                                                                                                                                                                                                                                                                                                                 |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                              |

## DATA21\_AC

|             |                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__data21</code> located constant data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA21\_AN

|             |                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__no_init __data21</code> located data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA21\_C

|                     |                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------|
| Description         | Holds <code>__data21</code> constant data. This can include constant variables, string and aggregate literals, etc. |
| Segment memory type | CONST                                                                                                               |
| Memory placement    | 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFF                                                                               |
| Access type         | Read-only                                                                                                           |

## DATA21\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__data21</code> static and global initialized variables initialized by copying from the segment <code>DATA21_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Memory placement    | 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFFF                                                                                                                                                                                                                                                                                                                                                                                                     |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## DATA21\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__data21</code> static and global variables in the <code>DATA21_I</code> segment. These values are copied from <code>DATA21_ID</code> to <code>DATA21_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFFF                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

## DATA21\_N

|                     |                                                                    |
|---------------------|--------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __data21</code> variables. |
| Segment memory type | DATA                                                               |
| Memory placement    | 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFFF                             |
| Access type         | Read-write                                                         |

## DATA21\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds zero-initialized <code>__data21</code> static and global variables. The contents of this segment is declared by the system startup code.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Memory placement    | 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFF                                                                                                                                                                                                                                                                                                                                                                                   |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                              |

## DATA32\_AC

|             |                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__data32</code> located constant data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA32\_AN

|             |                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__no_init __data32</code> located data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DATA32\_C

|                     |                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------|
| Description         | Holds <code>__data32</code> constant data. This can include constant variables, string and aggregate literals, etc. |
| Segment memory type | CONST                                                                                                               |
| Memory placement    | This segment can be placed anywhere in memory.                                                                      |
| Access type         | Read-only                                                                                                           |

## DATA32\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__data32</code> static and global initialized variables initialized by copying from the segment <code>DATA32_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                             |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## DATA32\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__data32</code> static and global variables in the <code>DATA32_I</code> segment. These values are copied from <code>DATA32_ID</code> to <code>DATA32_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CONST                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

## DATA32\_N

|                     |                                                                    |
|---------------------|--------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __data32</code> variables. |
| Segment memory type | DATA                                                               |
| Memory placement    | This segment can be placed anywhere in memory.                     |
| Access type         | Read-write                                                         |

## DATA32\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds zero-initialized <code>__data32</code> static and global variables. The contents of this segment is declared by the system startup code.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                                                                                                                                                                                                                                                          |
| Access type         | Read-write                                                                                                                                                                                                                                                                                                                                                                                                              |

## DBGREG\_AC

|             |                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__dbgreg</code> located constant data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DBGREG\_AN

|             |                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__no_init __dbgreg</code> located data.<br><br><i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## DIFUNCT

|                     |                                                      |
|---------------------|------------------------------------------------------|
| Description         | Holds the dynamic initialization vector used by C++. |
| Segment memory type | CONST                                                |
| Memory placement    | This segment can be placed anywhere in memory.       |
| Access type         | Read-only                                            |

## EVBYTES1

|                     |                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the actual code bytes of the <code>EVTAB</code> segment when linking for a segment translation system.                               |
| Segment memory type | <code>CODE</code>                                                                                                                          |
| Memory placement    | This segment must be placed at the same address as the offset of the <code>EVTAB</code> segment from the translation segment base address. |
| Access type         | Read-only                                                                                                                                  |
| See also            | <i>Linking for segment-translated systems</i> , page 114                                                                                   |

## EVBYTES2

|                     |                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the actual code bytes of the <code>EV100</code> segment when linking for a segment translation system.                               |
| Segment memory type | <code>CODE</code>                                                                                                                          |
| Memory placement    | This segment must be placed at the same address as the offset of the <code>EV100</code> segment from the translation segment base address. |
| Access type         | Read-only                                                                                                                                  |
| See also            | <i>Linking for segment-translated systems</i> , page 114                                                                                   |

## EVBYTES3

|                     |                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the actual code bytes of the <code>EVSEG</code> segment when linking for a segment translation system.                               |
| Segment memory type | <code>CODE</code>                                                                                                                          |
| Memory placement    | This segment must be placed at the same address as the offset of the <code>EVSEG</code> segment from the translation segment base address. |
| Access type         | Read-only                                                                                                                                  |
| See also            | <i>Linking for segment-translated systems</i> , page 114                                                                                   |



## EVSEG

|                     |                                                                                                                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds the code for exception handlers.</p> <p>When a system exception occurs, code located in this segment will be executed.</p>                                                  |
| Segment memory type | CODE                                                                                                                                                                                 |
| Memory placement    | For security reasons, this segment should be located in the privileged address space. Read more about how to locate the exception handlers in the chip manufacturer's documentation. |
| Access type         | Read-write                                                                                                                                                                           |

## EVTAB

|                     |                                                                                                                                                                                                                                                |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds code for exception handlers.</p> <p>The system register <i>EVBA</i> is automatically initialized to the start address for this segment at startup. When a system exception occurs, code located in this segment will be executed.</p> |
| Segment memory type | CODE                                                                                                                                                                                                                                           |
| Memory placement    | For security reasons, this segment should be located in the privileged address space. Read more about how to locate the exception handlers in the chip manufacturer's documentation.                                                           |
| Access type         | Read-write                                                                                                                                                                                                                                     |

## EVI00

|                     |                                                                                                                                                                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds code for exception handlers.</p> <p>When a system exception occurs, code located in this segment will be executed.</p>                                                                                                                                     |
| Segment memory type | CODE                                                                                                                                                                                                                                                                |
| Memory placement    | Should be placed at the address in the <i>EVBA</i> register + $0 \times 100$ . For security reasons, this segment should be located in the privileged address space. Read more about how to locate the exception handlers in the chip manufacturer's documentation. |
| Access type         | Read-write                                                                                                                                                                                                                                                          |

## FVVEC

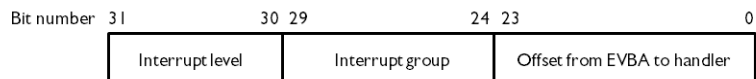
|                     |                                                                                                                                                                                                      |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds secure state exception vectors for FlashVault-enabled applications.</p> <p>Functions declared with the directive <code>#pragma flashvault_impl</code> generate entries in this segment.</p> |
| Segment memory type | CONST                                                                                                                                                                                                |
| Memory placement    | This segment must be placed at the start of the flash memory.                                                                                                                                        |
| Access type         | Read-only                                                                                                                                                                                            |
| See also            | <i>Implementing middleware using FlashVault™</i> , page 84                                                                                                                                           |

## HEAP

|                     |                                                                                                                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the heap used for dynamically allocated data, in other words data allocated by <code>malloc</code> and <code>free</code> , and in C++, <code>new</code> and <code>delete</code> . |
| Segment memory type | DATA                                                                                                                                                                                    |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                                          |
| Access type         | Read-write                                                                                                                                                                              |

## HTAB

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds information about how the interrupt controller should be initialized. The system startup code calls a function called <code>__init_ihandlers</code>, which in turn parses this segment and performs the requested initializations.</p> <p>The table consists of one 32-bit entry for each interrupt group for which a handler has been provided. The entries are not ordered in any special way and the format of each individual entry is as follows:</p> |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



|                     |                                                |
|---------------------|------------------------------------------------|
| Segment memory type | CONST                                          |
| Memory placement    | This segment can be placed anywhere in memory. |

|             |                                  |
|-------------|----------------------------------|
| Access type | Read-only                        |
| See also    | <i>System startup</i> , page 132 |

## INITTAB

|             |                                                                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds information about segments to be initialized with zero initial data (<code>_Z</code> segments), and segments to be copied during startup (<code>_ID</code> and <code>_I</code> segments).</p> <p>Each row in this table is contains three 32-bit parameters, corresponding to a declaration like:</p> |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

```

DC32      count
DC32      to
DC32      from

```

The first parameter (*count*) in each row always denotes the number of bytes to clear or copy.

The second parameter (*to*) contains the start address of a segment to initialize.

The third parameter (*from*) can contain the same value as the second parameter (*to*). In this case, the segment denoted by the second parameter will be cleared (set to zero). The cleared segment is one of the `_Z` segments.

If the third parameter (*from*) is different from the second parameter (*to*), it denotes the start of a segment to copy from. In this case, bytes will be copied from the “from” segment to the “to” segment. The “from” segment is one of the `_ID` segments, and the “to” segment is the corresponding `_I` segment.

**Note:** The contents of the `INITTAB` segment are automatically generated by the compiler.

|                     |                                                |
|---------------------|------------------------------------------------|
| Segment memory type | CONST                                          |
| Memory placement    | This segment can be placed anywhere in memory. |
| Access type         | Read-only                                      |

## RAMCODE2 I

|                     |                                                      |
|---------------------|------------------------------------------------------|
| Description         | Holds program code declared <code>__ramfunc</code> . |
| Segment memory type | DATA                                                 |

Memory placement 0x0-0xFFFFF and 0xFFF00000-0xFFFFFFFF

Access type Read-write

## RAMCODE21\_ID

Description Holds code for `__ramfunc` declared functions in the `RAMCODE21` segment. These values are copied from `RAMCODE21_ID` to `RAMCODE21` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be contiguous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.

Segment memory type `CONST`

Memory placement This segment can be placed anywhere in memory.

Access type Read-only

## RAMCODE32

Description Holds program code declared `__ramfunc`.

Segment memory type `DATA`

Memory placement This segment can be placed anywhere in memory.

Access type Read-write

## RAMCODE32\_ID

Description Holds code for `__ramfunc` declared functions in the `RAMCODE32` segment. These values are copied from `RAMCODE32_ID` to `RAMCODE32` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be contiguous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.

Segment memory type `CONST`

Memory placement This segment can be placed anywhere in memory.

|             |           |
|-------------|-----------|
| Access type | Read-only |
|-------------|-----------|

## RESET

|                     |                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------|
| Description         | Holds the code that is executed when the processor has been reset.                          |
| Segment memory type | CODE                                                                                        |
| Memory placement    | See the hardware reference manual for information about the location of the reset location. |
| Access type         | Read-only                                                                                   |

## RESETCODE

|                     |                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the actual code bytes of the <code>RESET</code> segment when linking for a segment translation system.                               |
| Segment memory type | CODE                                                                                                                                       |
| Memory placement    | This segment must be placed at the same address as the offset of the <code>RESET</code> segment from the translation segment base address. |
| Access type         | Read-only                                                                                                                                  |
| See also            | <i>Linking for segment-translated systems</i> , page 114.                                                                                  |

## SSTACK

|                     |                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the internal data and return stack in <i>supervisor</i> mode.                                                         |
| Segment memory type | DATA                                                                                                                        |
| Memory placement    | This segment can be placed anywhere in memory.                                                                              |
| Access type         | Read-write                                                                                                                  |
| See also            | For information about how to define this segment and its length in the linker command file, see <i>The stack</i> , page 68. |

## SWITCH

|                     |                                                     |
|---------------------|-----------------------------------------------------|
| Description         | Holds switch tables generated by switch statements. |
| Segment memory type | CONST                                               |
| Memory placement    | This segment can be placed anywhere in memory.      |
| Access type         | Read-only                                           |

## SYSREG\_AC

|             |                                                                                                                                                                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__sysreg</code> located constant data.<br><br><i>Located</i> means being placed at an absolute location using the @ operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## SYSREG\_AN

|             |                                                                                                                                                                                                                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__sysreg</code> located uninitialized data.<br><br><i>Located</i> means being placed at an absolute location using the @ operator or the <code>#pragma location</code> directive. Because the location is known, this segment does not need to be specified in the linker configuration file. |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## TRACEBUFFER

|                     |                                                |
|---------------------|------------------------------------------------|
| Description         | Holds trace data when using NanoTrace.         |
| Segment memory type | DATA                                           |
| Memory placement    | This segment can be placed anywhere in memory. |
| Access type         | No-access                                      |
| See also            | The <i>C-SPY® Debugging Guide for AVR32</i> .  |

# The stack usage control file

- Overview
- Stack usage control directives
- Syntactic components

Before you read this chapter, see *Stack usage analysis*, page 95.

---

## Overview

A stack usage control file consists of a sequence of directives that control stack usage analysis. You can use C ("*/\*...\*/*") and C++ ("*//...*") comments in these files.

The default filename extension for stack usage control files is `suc`.

### C++ NAMES

You can also use wildcards in function names. "`#*`" matches any sequence of characters, and "`#?`" matches a single character.

---

## Stack usage control directives

This section gives detailed reference information about each stack usage control directive.

### call graph root directive

|             |                                                                                                                                                                                                                             |                                 |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| Syntax      | <code>call graph root [ <i>category</i> ] : <i>func-spec</i> [ , <i>func-spec</i>... ] ;</code>                                                                                                                             |                                 |
| Parameters  | <i>category</i>                                                                                                                                                                                                             | See <i>category</i> , page 387  |
|             | <i>func-spec</i>                                                                                                                                                                                                            | See <i>func-spec</i> , page 387 |
| Description | Specifies that the listed functions are call graph roots. You can optionally specify a call graph root category. Call graph roots are listed under their category in the <i>Stack Usage</i> chapter in the linker map file. |                                 |

The linker will normally issue a warning for functions needed in the application that are not call graph roots and which do not appear to be called.

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

**See also** `call_graph_root`, page 308.

## check that directive

**Syntax** `check that expression;`

**Parameters** `expression` A boolean expression.

**Description** You can use the `check that` directive to compare the results of stack usage analysis against the sizes of blocks and regions. If the expression evaluates to zero, an error is emitted.

Three extra operators are available for use only in `check that` expressions:

`maxstack(category)` The stack depth of the deepest call chain for any call graph root function in the category.

`totalstack(category)` The sum of the stack depths of the deepest call chains for each call graph root function in the category.

`size("SEGMENT")` The size of the segment.

**Example** `check that maxstack("Program entry")  
+ totalstack("interrupt")  
+ 1K  
<= size("CSTACK");`

**See also** `Stack usage analysis`, page 95.

## exclude directive

**Syntax** `exclude func-spec [, func-spec... ];`

**Parameters** `func-spec` See `func-spec`, page 387



|             |                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------|
| Description | Excludes the specified functions, and call trees originating with them, from stack usage calculations. |
| Example     | <code>exclude MyFunc5, MyFunc6;</code>                                                                 |

## function directive

|                   |                                                                                                                                                                                                                                                                                                                                                                                                          |                 |                                |                  |                                 |                  |                                 |                   |                                  |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|--------------------------------|------------------|---------------------------------|------------------|---------------------------------|-------------------|----------------------------------|
| Syntax            | <code>[ <i>override</i> ] function [ <i>category</i> ] <i>func-spec</i> : <i>stack-size</i><br/>[ , <i>call-info</i>... ];</code>                                                                                                                                                                                                                                                                        |                 |                                |                  |                                 |                  |                                 |                   |                                  |
| Parameters        | <table> <tr> <td><i>category</i></td> <td>See <i>category</i>, page 387</td> </tr> <tr> <td><i>func-spec</i></td> <td>See <i>func-spec</i>, page 387</td> </tr> <tr> <td><i>call-info</i></td> <td>See <i>call-info</i>, page 388</td> </tr> <tr> <td><i>stack-size</i></td> <td>See <i>stack-size</i>, page 389</td> </tr> </table>                                                                     | <i>category</i> | See <i>category</i> , page 387 | <i>func-spec</i> | See <i>func-spec</i> , page 387 | <i>call-info</i> | See <i>call-info</i> , page 388 | <i>stack-size</i> | See <i>stack-size</i> , page 389 |
| <i>category</i>   | See <i>category</i> , page 387                                                                                                                                                                                                                                                                                                                                                                           |                 |                                |                  |                                 |                  |                                 |                   |                                  |
| <i>func-spec</i>  | See <i>func-spec</i> , page 387                                                                                                                                                                                                                                                                                                                                                                          |                 |                                |                  |                                 |                  |                                 |                   |                                  |
| <i>call-info</i>  | See <i>call-info</i> , page 388                                                                                                                                                                                                                                                                                                                                                                          |                 |                                |                  |                                 |                  |                                 |                   |                                  |
| <i>stack-size</i> | See <i>stack-size</i> , page 389                                                                                                                                                                                                                                                                                                                                                                         |                 |                                |                  |                                 |                  |                                 |                   |                                  |
| Description       | <p>Specifies what the maximum stack usage is in a function and which other functions that are called from that function.</p> <p>Normally, an error is issued if there already is stack usage information for the function, but if you start with <code>override</code>, the error will be suppressed and the information supplied in the directive will be used instead of the previous information.</p> |                 |                                |                  |                                 |                  |                                 |                   |                                  |
| Example           | <pre>function MyFunc1: 32,     calls MyFunc2,     calls MyFunc3, MyFunc4: 16;  function [interrupt] MyInterruptHandler: 44;</pre>                                                                                                                                                                                                                                                                        |                 |                                |                  |                                 |                  |                                 |                   |                                  |

## max recursion depth directive

|                  |                                                                                                                                                                  |                  |                                 |             |                            |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------------------------------|-------------|----------------------------|
| Syntax           | <code>max recursion depth <i>func-spec</i> : <i>size</i>;</code>                                                                                                 |                  |                                 |             |                            |
| Parameters       | <table> <tr> <td><i>func-spec</i></td> <td>See <i>func-spec</i>, page 387</td> </tr> <tr> <td><i>size</i></td> <td>See <i>size</i>, page 389</td> </tr> </table> | <i>func-spec</i> | See <i>func-spec</i> , page 387 | <i>size</i> | See <i>size</i> , page 389 |
| <i>func-spec</i> | See <i>func-spec</i> , page 387                                                                                                                                  |                  |                                 |             |                            |
| <i>size</i>      | See <i>size</i> , page 389                                                                                                                                       |                  |                                 |             |                            |
| Description      | Specifies the maximum number of iterations through any of the cycles in the recursion nest of which the function is a member.                                    |                  |                                 |             |                            |

A recursion nest is a set of cycles in the call graph where each cycle shares at least one node with another cycle in the nest.

Stack usage analysis will base its result on the max recursion depth multiplied by the stack usage of the deepest cycle in the nest. If the nest is not entered on a point along one of the deepest cycles, no stack usage result will be calculated for such calls.

#### Example

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

## no calls from directive

#### Syntax

```
no calls from module-spec to func-spec [ , func-spec... ];
```

#### Parameters

*func-spec*

See *func-spec*, page 387

*module-spec*

See *module-spec*, page 387

#### Description

When you provide stack usage information for some functions in a module without stack usage information, the linker warns about functions that are referenced from the module but not listed as called. This is primarily to help avoid problems with C runtime routines, calls to which are generated by the compiler, beyond user control.

If there actually is no call to some of these functions, use the `no calls from` directive to selectively suppress the warning for the specified functions. You can also disable the warning entirely (`--diag_suppress` or

**Project>Options>Linker>Diagnostics>Suppress these diagnostics**).

#### Example

```
no calls from [file.r82] to MyFunc13, MyFun14;
```

## possible calls directive

#### Syntax

```
possible calls calling-func : called-func [ , called-func... ];
```

#### Parameters

*calling-func*

See *func-spec*, page 387

*called-func*

See *func-spec*, page 387

#### Description

Specifies an exhaustive list of possible destinations for all indirect calls in one function. Use this for functions which are known to perform indirect calls and where you know exactly which functions that might be called in this particular application. Consider

using the `#pragma calls` directive if the information about which functions that might be called is available when compiling.

|          |                                                        |
|----------|--------------------------------------------------------|
| Example  | <code>possible calls MyFunc7: MyFunc8, MyFunc9;</code> |
| See also | <i>calls</i> , page 308.                               |

---

## Syntactic components

The stack usage control directives use some syntactical components. These are described below.

### ***category***

|             |                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------|
| Syntax      | <code>[ <i>name</i> ]</code>                                                                  |
| Description | A call graph root category. You can use any name you like. Categories are not case-sensitive. |
| Example     | category examples:<br><br><code>[interrupt]</code><br><code>[task]</code>                     |

### ***func-spec***

|             |                                                                                                                                                                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>[ ? ] <i>name</i> [ <i>module-spec</i> ]</code>                                                                                                                                                                                                     |
| Description | Specifies the name of a symbol, and for module-local symbols, the name of the module it is defined in. Normally, if <i>func-spec</i> does not match a symbol in the program, a warning is emitted. Prefixing with <code>?</code> suppresses this warning. |
| Example     | <i>func-spec</i> examples:<br><br><code>xFun</code><br><code>MyFun [file.r82]</code>                                                                                                                                                                      |

### ***module-spec***

|        |                                       |
|--------|---------------------------------------|
| Syntax | <code>[name [ (<i>name</i>) ]]</code> |
|--------|---------------------------------------|

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Specifies the name of a module, and optionally, in parentheses, the name of the library it belongs to. To distinguish between modules with the same name, you can specify:</p> <ul style="list-style-type: none"> <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> <p>Note that when using multi-file compilation (<code>-mfc</code>), multiple files are compiled into a single module, named after the first file.</p> |
| Example     | <p><i>module-spec</i> examples:</p> <pre>[file.r82] [file.r82(lib.a)] ["D:\C1\test\file.r82"]</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

## ***name***

|             |                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>A name can be either an identifier or a quoted string.</p> <p>The first character of an identifier must be either a letter or one of the characters "_", "\$", or ".". The rest of the characters can also be digits.</p> <p>A quoted string starts and ends with " and can contain any character. Two consecutive " characters can be used inside a quoted string to represent a single ".</p> |
| Example     | <p><i>name</i> examples:</p> <pre>MyFun file.r82 "file-1.r82"</pre>                                                                                                                                                                                                                                                                                                                                |

## ***call-info***

|             |                                                                                                       |
|-------------|-------------------------------------------------------------------------------------------------------|
| Syntax      | <code>calls <i>func-spec</i> [ , <i>func-spec</i>... ] [ : <i>stack-size</i> ]</code>                 |
| Description | Specifies one or more called functions, and optionally, the stack size at the calls.                  |
| Example     | <p><i>call-info</i> examples:</p> <pre>calls MyFunc1 : stack 16 calls MyFunc2, MyFunc3, MyFunc4</pre> |

**stack-size**

|             |                                                            |
|-------------|------------------------------------------------------------|
| Syntax      | <code>[ stack ] size</code>                                |
| Description | Specifies the size of a stack frame.                       |
| Example     | <i>stack-size</i> examples:<br>24<br><code>stack 28</code> |

**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     | <i>size</i> examples:<br>24<br>0x18<br>2048<br>2K                                                                                                                                                                   |



# Implementation-defined behavior for Standard C

- Descriptions of implementation-defined behavior

If you are using C89 instead of Standard C, see *Implementation-defined behavior for C89*, page 407. For a short overview of the differences between Standard C and C89, see *C language overview*, page 187.

---

## Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

**Note:** The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

### J.3.1 TRANSLATION

#### Diagnostics (3.10, 5.1.1.3)

Diagnostics are produced in the form:

```
filename, linenumber level[tag): message
```

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

#### White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.

## J.3.2 ENVIRONMENT

### The character set (5.1.1.2)

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, if you use the `--enable_multibytes` compiler option, the host character set is used instead.

### Main (5.1.2.1)

The function called at program startup is called `main`. No prototype is declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior, see *Customizing system initialization*, page 135.

### The effect of program termination (5.1.2.1)

Terminating the application returns the execution to the startup code (just after the call to `main`).

### Alternative ways to define main (5.1.2.2.1)

There is no alternative ways to define the `main` function.

### The argv argument to main (5.1.2.2.1)

The `argv` argument is not supported.

### Streams as interactive devices (5.1.2.3)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

## Signals, their semantics, and the default handling (7.14)

In the DLIB library, the set of supported signals is the same as in Standard C. A raised signal will do nothing, unless the `signal` function is customized to fit the application.

### Signal values for computational exceptions (7.14.1.1)

In the DLIB library, there are no implementation-defined values that correspond to a computational exception.

### Signals at system startup (7.14.1.1)

In the DLIB library, there are no implementation-defined signals that are executed at system startup.



### **Environment names (7.20.4.5)**

In the DLIB library, there are no implementation-defined environment names that are used by the `getenv` function.

### **The system function (7.20.4.6)**

The `system` function is not supported.

## **J.3.3 IDENTIFIERS**

### **Multibyte characters in identifiers (6.4.2)**

Additional multibyte characters may not appear in identifiers.

### **Significant characters in identifiers (5.2.4.1, 6.1.2)**

The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

## **J.3.4 CHARACTERS**

### **Number of bits in a byte (3.6)**

A byte contains 8 bits.

### **Execution character set member values (5.2.1)**

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the host character set.

### **Alphabetic escape sequences (5.2.2)**

The standard alphabetic escape sequences have the values `\a-7`, `\b-8`, `\f-12`, `\n-10`, `\r-13`, `\t-9`, and `\v-11`.

### **Characters outside of the basic executive character set (6.2.5)**

A character outside of the basic executive character set that is stored in a `char` is not transformed.

### **Plain char (6.2.5, 6.3.1.1)**

A plain `char` is treated as an `unsigned char`.

### **Source and execution character sets (6.4.4.4, 5.1.1.2)**

The source character set is the set of legal characters that can appear in source files. By default, the source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. By default, the execution character set is the standard ASCII character set.

However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 141.

### **Integer character constants with more than one character (6.4.4.4)**

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

### **Wide character constants with more than one character (6.4.4.4)**

A wide character constant that contains more than one multibyte character generates a diagnostic message.

### **Locale used for wide character constants (6.4.4.4)**

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

### **Locale used for wide string literals (6.4.5)**

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

### **Source characters as executive characters (6.4.5)**

All source characters can be represented as executive characters.

## **J.3.5 INTEGERS**

### **Extended integer types (6.2.5)**

There are no extended integer types.

**Range of integer values (6.2.6.2)**

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

For information about the ranges for the different integer types, see *Basic data types—integer types*, page 276.

**The rank of extended integer types (6.3.1.1)**

There are no extended integer types.

**Signals when converting to a signed integer type (6.3.1.3)**

No signal is raised when an integer is converted to a signed integer type.

**Signed bitwise operations (6.5)**

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

**J.3.6 FLOATING POINT****Accuracy of floating-point operations (5.2.4.2.2)**

The accuracy of floating-point operations is unknown.

**Rounding behaviors (5.2.4.2.2)**

There are no non-standard values of `FLT_ROUNDS`.

**Evaluation methods (5.2.4.2.2)**

There are no non-standard values of `FLT_EVAL_METHOD`.

**Converting integer values to floating-point values (6.3.1.4)**

When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

**Converting floating-point values to floating-point values (6.3.1.5)**

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Denoting the value of floating-point constants (6.4.4.2)**

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Contraction of floating-point values (6.5)**

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

### **Default state of `FENV_ACCESS` (7.6.1)**

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

### **Additional floating-point mechanisms (7.6, 7.12)**

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

### **Default state of `FP_CONTRACT` (7.12.2)**

The default state of the pragma directive `FP_CONTRACT` is `OFF`.

## **J.3.7 ARRAYS AND POINTERS**

### **Conversion from/to pointers (6.3.2.3)**

For information about casting of data pointers and function pointers, see *Casting*, page 282.

### **`ptrdiff_t` (6.5.6)**

For information about `ptrdiff_t`, see *ptrdiff\_t*, page 282.

## **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 87.

## 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 277.

### 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 254.

### 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 277.

### 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 275.

### 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 284.

## J.3.11 PREPROCESSING DIRECTIVES

### Mapping of header names (6.4.7)

Sequences in header names are mapped to source file names verbatim. A backslash `'\'` is not treated as an escape sequence. See *Overview of the preprocessor*, page 347.

### 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 246.

### 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 232.

### 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 232.

### Preprocessing tokens in #include directives (6.10.2)

Preprocessing tokens in an `#include` directive are combined in the same way as outside an `#include` directive.

### Nesting limits for #include directives (6.10.2)

There is no explicit nesting limit for `#include` processing.

### Universal character names (6.10.3.2)

Universal character names (UCN) are not supported.

### Recognized pragma directives (6.10.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the chapter *Pragma directives* and here, the information provided in the chapter *Pragma directives* overrides the information here.

```
alignment
baseaddr
basic_template_matching
building_runtime
can_instantiate
```

codeseg  
cspy\_support  
define\_type\_info  
do\_not\_instantiate  
early\_dynamic\_initialization  
function  
function\_effects  
hdrstop  
important\_typedef  
instantiate  
keep\_definition  
library\_default\_requirements  
library\_provides  
library\_requirement\_override  
memory  
module\_name  
no\_pch  
once  
system\_include  
warnings

### **Default `__DATE__` and `__TIME__` (6.10.8)**

The definitions for `__TIME__` and `__DATE__` are always available.

## **J.3.12 LIBRARY FUNCTIONS**

### **Additional library facilities (5.1.2.1)**

Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 119.

### **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 *fenv.h*, page 361.

### **Feraiseexcept raising floating-point exception (7.6.2.3)**

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 280.

### **Strings passed to the setlocale function (7.11.1.1)**

For information about strings passed to the `setlocale` function, see *Locale*, page 141.

### **Types defined for float\_t and double\_t (7.12)**

The `FLT_EVAL_METHOD` macro can only have the value 0.

### **Domain errors (7.12.1)**

No function generates other domain errors than what the standard requires.

### **Return values on domain errors (7.12.1)**

Mathematic functions return a floating-point NaN (not a number) for domain errors.

### **Underflow errors (7.12.1)**

Mathematic functions set `errno` to the macro `ERANGE` (a macro in `errno.h`) and return zero for underflow errors.

### **fmod return value (7.12.10.1)**

The `fmod` function returns a floating-point NaN when the second argument is zero.

### **The magnitude of remquo (7.12.10.3)**

The magnitude is congruent modulo `INT_MAX`.

### **signal() (7.14.1.1)**

The signal part of the library is not supported.



**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 145.

### **NULL macro (7.17)**

The `NULL` macro is defined to 0.

### **Terminating newline character (7.19.2)**

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

### **Space characters before a newline character (7.19.2)**

Space characters written to a stream immediately before a newline character are preserved.

### **Null characters appended to data written to binary streams (7.19.2)**

No null characters are appended to data written to binary streams.

### **File position in append mode (7.19.3)**

The file position is initially placed at the beginning of the file when it is opened in `append-mode`.

### **Truncation of files (7.19.3)**

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 141.

### **File buffering (7.19.3)**

An open file can be either block-buffered, line-buffered, or unbuffered.

### **A zero-length file (7.19.3)**

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

### **Legal file names (7.19.3)**

The legality of a filename depends on the application-specific implementation of the low-level file routines.

### **Number of times a file can be opened (7.19.3)**

Whether a file can be opened more than once depends on the application-specific implementation of the low-level file routines.

### **Multibyte characters in a file (7.19.3)**

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

### **remove() (7.19.4.1)**

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 141.

### **rename() (7.19.4.2)**

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 141.

### **Removal of open temporary files (7.19.4.3)**

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

### **Mode changing (7.19.5.4)**

`freopen` closes the named stream, then reopens it in the new mode. The streams `stdin`, `stdout`, and `stderr` can be reopened in any new mode.

### **Style for printing infinity or NaN (7.19.6.1, 7.24.2.1)**

The style used for printing infinity or NaN for a floating-point constant is `inf` and `nan` (`INF` and `NAN` for the `F` conversion specifier), respectively. The `n-char-sequence` is not used for `nan`.

### **%p in printf() (7.19.6.1, 7.24.2.1)**

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

### **Reading ranges in scanf (7.19.6.2, 7.24.2.1)**

A `-` (dash) character is always treated as a range symbol.

**%p in scanf (7.19.6.2, 7.24.2.2)**

The %p conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

**File position errors (7.19.9.1, 7.19.9.3, 7.19.9.4)**

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

**An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)**

An n-char-sequence after a NaN is read and ignored.

**errno value at underflow (7.20.1.3, 7.24.4.1.1)**

`errno` is set to `ERANGE` if an underflow is encountered.

**Zero-sized heap objects (7.20.3)**

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

**Behavior of abort and exit (7.20.4.1, 7.20.4.4)**

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

**Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)**

The termination status will be propagated to `__exit()` as a parameter. `exit()` and `_Exit()` use the input parameter, whereas `abort` uses `EXIT_FAILURE`.

**The system function return value (7.20.4.6)**

The `system` function is not supported.

**The time zone (7.23.1)**

The local time zone and daylight savings time must be defined by the application. For more information, see *Time*, page 145.

**Range and precision of time (7.23)**

For information about range and precision, see *time.h*, page 362. The application must supply the actual implementation for the functions `time` and `clock`. See *Time*, page 145.

### **clock() (7.23.2.1)**

The application must supply an implementation of the `clock` function. See *Time*, page 145.

### **%Z replacement string (7.23.3.5, 7.24.5.1)**

By default, ":" is used as a replacement for %Z. Your application should implement the time zone handling. See *Time*, page 145.

### **Math functions rounding mode (F.9)**

The functions in `math.h` honor the rounding direction mode in `FLT-ROUNDS`.

## **J.3.13 ARCHITECTURE**

### **Values and expressions assigned to some macros (5.2.4.2, 7.18.2, 7.18.3)**

There are always 8 bits in a byte.

`MB_LEN_MAX` is at the most 6 bytes depending on the library configuration that is used.

For information about sizes, ranges, etc for all basic types, see *Data representation*, page 275.

The limit macros for the exact-width, minimum-width, and fastest minimum-width integer types defined in `stdint.h` have the same ranges as `char`, `short`, `int`, `long`, and `long long`.

The floating-point constant `FLT_ROUNDS` has the value 1 (to nearest) and the floating-point constant `FLT_EVAL_METHOD` has the value 0 (treat as is).

### **The number, order, and encoding of bytes (6.2.6.1)**

See *Data representation*, page 275.

### **The value of the result of the sizeof operator (6.5.3.4)**

See *Data representation*, page 275.

## **J.4 LOCALE**

### **Members of the source and execution character set (5.2.1)**

By default, the compiler accepts all one-byte characters in the host's default character set. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

### **The meaning of the additional character set (5.2.1.2)**

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

### **Shift states for encoding multibyte characters (5.2.1.2)**

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

### **Direction of successive printing characters (5.2.2)**

The application defines the characteristics of a display device.

### **The decimal point character (7.1.1)**

The default decimal-point character is a '.'. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

### **Printing characters (7.4, 7.25.2)**

The set of printing characters is determined by the chosen locale.

### **Control characters (7.4, 7.25.2)**

The set of control characters is determined by the chosen locale.

### **Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)**

The sets of characters tested are determined by the chosen locale.

### **The native environment (7.1.1.1)**

The native environment is the same as the "C" locale.

### **Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)**

There are no additional subject sequences that can be accepted by the numeric conversion functions.

### **The collation of the execution character set (7.21.4.3, 7.24.4.4.2)**

The collation of the execution character set is determined by the chosen locale.

**Message returned by strerror (7.21.6.2)**

The messages returned by the `strerror` function depending on the argument is:

| <b>Argument</b> | <b>Message</b>            |
|-----------------|---------------------------|
| 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 47: Message returned by strerror()—IAR DLIB library*

# 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 391. For a short overview of the differences between Standard C and C89, see *C language overview*, page 187.

---

## 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 IAR DLIB runtime environment, see *Customizing system initialization*, page 135.

### **Interactive devices (5.1.2.3)**

The streams `stdin` and `stdout` are treated as interactive devices.

## **IDENTIFIERS**

### **Significant characters without external linkage (6.1.2)**

The number of significant initial characters in an identifier without external linkage is 200.

### **Significant characters with external linkage (6.1.2)**

The number of significant initial characters in an identifier with external linkage is 200.

### **Case distinctions are significant (6.1.2)**

Identifiers with external linkage are treated as case-sensitive.

## **CHARACTERS**

### **Source and execution character sets (5.2.1)**

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 141.

### **Bits per character in execution character set (5.2.4.2.1)**

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

### **Mapping of characters (6.1.3.4)**

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.



### Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

### Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

### Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 141.

### 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 276, for information about the ranges for the different integer types.

### Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

**Signed bitwise operations (6.3)**

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

**Sign of the remainder on integer division (6.3.5)**

The sign of the remainder on integer division is the same as the sign of the dividend.

**Negative valued signed right shifts (6.3.7)**

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

**FLOATING POINT****Representation of floating-point values (6.1.2.5)**

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (*s*), a biased exponent (*e*), and a mantissa (*m*).

See *Basic data types—floating-point types*, page 279, 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 282, for information about `size_t`.

**Conversion from/to pointers (6.3.4)**

See *Casting*, page 282, for information about casting of data pointers and function pointers.

**ptrdiff\_t (6.3.6, 7.1.1)**

See *ptrdiff\_t*, page 282, 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 276, 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
basic_template_matching
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
function_effects
hdrstop
important_typedef
instantiate
keep_definition
library_default_requirements
library_provides
library_requirement_override
memory
module_name
no_pch
once
system_include
warnings
```

**Default `__DATE__` and `__TIME__` (6.8.8)**

The definitions for `__TIME__` and `__DATE__` are always available.

**IAR DLIB LIBRARY FUNCTIONS**

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

**NULL macro (7.1.6)**

The `NULL` macro is defined to 0.

**Diagnostic printed by the `assert` function (7.2)**

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

**Domain errors (7.5.1)**

NaN (Not a Number) will be returned by the mathematic functions on domain errors.

**Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)**

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

**`fmod()` functionality (7.5.6.4)**

If the second argument to `fmod()` is zero, the function returns NaN; `errno` is set to `EDOM`.

**`signal()` (7.7.1.1)**

The signal part of the library is not supported.

**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 145.

**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 *File input and output*, page 141.

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 *File input and output*, page 141.

**rename() (7.9.4.2)**

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 141.

**%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 *Environment interaction*, page 144.

### **system() (7.10.4.5)**

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 144.



### 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 48: Message returned by `strerror()`—IAR DLIB library

### The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 145.

#### `clock()` (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 145.



## A

- ABI . . . . . 171
- abort
  - implementation-defined behavior . . . . . 403
  - implementation-defined behavior in C89 (DLIB) . . . . . 416
  - system termination (DLIB) . . . . . 134
- absolute location
  - data, placing at (@) . . . . . 215
  - language support for . . . . . 190
  - #pragma location . . . . . 318
- abs\_s (ETSI macro) . . . . . 331
- \_\_acall (extended keyword) . . . . . 294
- acall functions . . . . . 77
- ACALL jump table . . . . . 111
- ACTAB (segment) . . . . . 367
- add (ETSI macro) . . . . . 331
- addressing. *See* memory types, data models, and code models
- algorithm (STL header file) . . . . . 359
- alignment . . . . . 275
  - forcing stricter (#pragma data\_alignment) . . . . . 309
  - in structures (#pragma pack) . . . . . 321
  - in structures, causing problems . . . . . 212
  - of an object (\_\_ALIGNOF\_\_) . . . . . 190
  - restrictions for inline assembler . . . . . 161
- alignment (pragma directive) . . . . . 398, 413
- \_\_ALIGNOF\_\_ (operator) . . . . . 190
- anonymous structures . . . . . 213
- anonymous symbols, creating . . . . . 187
- ANSI C. *See* C89
- application
  - building, overview of . . . . . 54
  - execution, overview of . . . . . 50
  - startup and termination (DLIB) . . . . . 131
- ARGFRAME (assembler directive) . . . . . 178
- argv (argument), implementation-defined behavior . . . . . 392
- arrays
  - designated initializers in . . . . . 187
  - global, accessing . . . . . 180
  - hints about index type . . . . . 211
  - implementation-defined behavior . . . . . 396
  - implementation-defined behavior in C89 . . . . . 410
  - incomplete at end of structs . . . . . 187
  - non-lvalue . . . . . 193
  - of incomplete types . . . . . 192
  - single-value initialization . . . . . 193
- asm, \_\_asm (language extension) . . . . . 162
- assembler code
  - calling from C . . . . . 167
  - calling from C++ . . . . . 169
  - inserting inline . . . . . 160
- assembler directives
  - for call frame information . . . . . 182
  - for static overlay . . . . . 178
  - using in inline assembler code . . . . . 161
- assembler instructions, inserting inline . . . . . 160
- assembler labels
  - default for application startup . . . . . 54
  - making public (--public\_equ) . . . . . 268
- assembler language interface . . . . . 159
  - calling convention. *See* assembler code
- assembler list file, generating . . . . . 258
- assembler output file . . . . . 169
- asserts . . . . . 148
  - implementation-defined behavior of . . . . . 400
  - implementation-defined behavior of in C89, (DLIB) . . . . . 414
  - including in application . . . . . 353
- assert.h (DLIB header file) . . . . . 357
- \_\_assignment\_by\_bitwise\_copy\_allowed, symbol used in library . . . . . 362
- @ (operator)
  - placing at absolute address . . . . . 215
  - placing in segments . . . . . 217
- atomic operations . . . . . 79
  - \_\_monitor . . . . . 298
- attributes
  - object . . . . . 292
  - type . . . . . 289

|                                             |     |
|---------------------------------------------|-----|
| auto variables                              | 68  |
| at function entrance                        | 173 |
| programming hints for efficient code        | 224 |
| using in inline assembler statements        | 161 |
| --avr32_dsp_instructions (compiler option)  | 243 |
| --avr32_flashvault (compiler option)        | 243 |
| --avr32_fpu_instructions (compiler option)  | 244 |
| --avr32_rmw_instructions (compiler option)  | 244 |
| --avr32_simd_instructions (compiler option) | 245 |

## B

|                                                    |                                   |
|----------------------------------------------------|-----------------------------------|
| backtrace information                              | <i>See</i> call frame information |
| Barr, Michael                                      | 33                                |
| baseaddr (pragma directive)                        | 398, 413                          |
| __BASE_FILE__ (predefined symbol)                  | 348                               |
| basic_template_matching (pragma directive)         | 398, 413                          |
| batch files                                        |                                   |
| error return codes                                 | 234                               |
| none for building library from command line        | 130                               |
| binary streams                                     | 401                               |
| binary streams in C89 (DLIB)                       | 415                               |
| bit negation                                       | 225                               |
| bitfields                                          |                                   |
| data representation of                             | 277                               |
| implementation-defined behavior                    | 397                               |
| implementation-defined behavior in C89             | 411                               |
| non-standard types in                              | 190                               |
| bitfields (pragma directive)                       | 307                               |
| bits in a byte, implementation-defined behavior    | 393                               |
| __bit_reverse (intrinsic function)                 | 333                               |
| bit, efficient manipulation using RMW instructions | 181                               |
| bold style, in this guide                          | 34                                |
| bool (data type)                                   | 276                               |
| adding support for in DLIB                         | 358, 360                          |
| __BREAKPOINT (intrinsic function)                  | 333                               |
| building_runtime (pragma directive)                | 398, 413                          |
| __BUILD_NUMBER__ (predefined symbol)               | 348                               |

## C

|                                                 |                               |
|-------------------------------------------------|-------------------------------|
| C and C++ linkage                               | 171                           |
| C/C++ calling convention                        | <i>See</i> calling convention |
| C header files                                  | 357                           |
| C language, overview                            | 187                           |
| __cache_control (intrinsic function)            | 334                           |
| call frame information                          | 181                           |
| in assembler list file                          | 169                           |
| in assembler list file (-IA)                    | 258                           |
| call graph root (stack usage control directive) | 383                           |
| call stack                                      | 181                           |
| callee-save registers, stored on stack          | 68                            |
| calling convention                              |                               |
| C++, requiring C linkage                        | 169                           |
| in compiler                                     | 170                           |
| calloc (library function)                       | 69                            |
| <i>See also</i> heap                            |                               |
| implementation-defined behavior in C89 (DLIB)   | 416                           |
| calls (pragma directive)                        | 308                           |
| call_graph_root (pragma directive)              | 308                           |
| call-info (in stack usage control file)         | 388                           |
| can_instantiate (pragma directive)              | 398, 413                      |
| cassert (library header file)                   | 360                           |
| cast operators                                  |                               |
| in Extended EC++                                | 196, 200                      |
| missing from Embedded C++                       | 196                           |
| casting                                         |                               |
| of pointers and integers                        | 282                           |
| pointers to integers, language extension        | 192                           |
| category (in stack usage control file)          | 387                           |
| cctype (DLIB header file)                       | 360                           |
| cerrno (DLIB header file)                       | 360                           |
| cexit (system termination code)                 |                               |
| customizing system termination                  | 135                           |
| in DLIB                                         | 132                           |
| CFI (assembler directive)                       | 182                           |
| cfloat (DLIB header file)                       | 360                           |

- char (data type) . . . . . 276
  - changing default representation (`--char_is_signed`) . . . 246
  - changing representation (`--char_is_unsigned`) . . . . . 246
  - implementation-defined behavior. . . . . 393
  - signed and unsigned. . . . . 277
- character set, implementation-defined behavior . . . . . 392
- characters
  - implementation-defined behavior. . . . . 393
  - implementation-defined behavior in C89. . . . . 408
- character-based I/O. . . . . 137
- `--char_is_signed` (compiler option) . . . . . 246
- `--char_is_unsigned` (compiler option) . . . . . 246
- check that (linker directive). . . . . 384
- checksum
  - calculation of. . . . . 207
- CHECKSUM (segment) . . . . . 368
- cinttypes (DLIB header file) . . . . . 360
- `__clear_status_flag` (intrinsic function). . . . . 334
- climits (DLIB header file). . . . . 360
- clobber . . . . . 161
- locale (DLIB header file) . . . . . 360
- clock (DLIB library function),  
implementation-defined behavior in C89 . . . . . 417
- clock (library function)
- implementation-defined behavior . . . . . 404
- clock.c . . . . . 145
- `__close` (DLIB library function) . . . . . 141
- clustering (compiler transformation). . . . . 223
  - disabling (`--no_clustering`). . . . . 261
- cmain (system initialization code)
  - in DLIB . . . . . 132
- cmath (DLIB header file) . . . . . 360
- code
  - execution of . . . . . 56
  - facilitating for good generation of . . . . . 223
  - interruption of execution . . . . . 73
  - verifying linked result . . . . . 117
- code models . . . . . 71
  - calling functions in. . . . . 178
  - configuration . . . . . 56
  - identifying (`__CODE_MODEL__`) . . . . . 348
  - selecting (`--code_model`) . . . . . 246
- code motion (compiler transformation). . . . . 222
  - disabling (`--no_code_motion`) . . . . . 261
- codeseg (pragma directive) . . . . . 399, 413
- `__CODE_MODEL__` (predefined symbol). . . . . 348
- `__code_model` (runtime model attribute) . . . . . 156
- `--code_model` (compiler option) . . . . . 246
- `__code`, symbol used in library . . . . . 362
- `__code21` (extended keyword) . . . . . 294
- CODE21 (segment). . . . . 368
- `__code32` (extended keyword) . . . . . 294
- CODE32 (segment). . . . . 368
- command line options
  - See also* compiler options
  - part of compiler invocation syntax . . . . . 231
  - passing. . . . . 232
  - typographic convention . . . . . 34
- command prompt icon, in this guide. . . . . 34
- comments
  - after preprocessor directives. . . . . 193
  - C++ style, using in C code. . . . . 187
- common block (call frame information) . . . . . 182
- common subexpr elimination (compiler transformation) . 221
  - disabling (`--no_cse`) . . . . . 261
- compilation date
  - exact time of (`__TIME__`) . . . . . 352
  - identifying (`__DATE__`) . . . . . 349
- compiler
  - environment variables . . . . . 232
  - invocation syntax . . . . . 231
  - output from . . . . . 233
- compiler listing, generating (-l). . . . . 258
- compiler object file . . . . . 47
  - including debug information in (`--debug, -r`) . . . . . 249
  - output from compiler . . . . . 233
- compiler optimization levels . . . . . 220
- compiler options . . . . . 237
  - passing to compiler . . . . . 232

|                                                                             |          |
|-----------------------------------------------------------------------------|----------|
| reading from file (-f) . . . . .                                            | 256      |
| specifying parameters . . . . .                                             | 239      |
| summary . . . . .                                                           | 239      |
| syntax. . . . .                                                             | 237      |
| for creating skeleton code . . . . .                                        | 168      |
| instruction scheduling . . . . .                                            | 223      |
| --warnings_affect_exit_code . . . . .                                       | 234      |
| compiler platform, identifying . . . . .                                    | 351      |
| compiler subversion number. . . . .                                         | 352      |
| compiler transformations . . . . .                                          | 218      |
| compiler version number . . . . .                                           | 353      |
| compiling                                                                   |          |
| from the command line . . . . .                                             | 54       |
| syntax. . . . .                                                             | 231      |
| complex numbers, supported in Embedded C++. . . . .                         | 196      |
| complex (library header file). . . . .                                      | 359      |
| complex.h (library header file) . . . . .                                   | 357      |
| compound literals . . . . .                                                 | 187      |
| computer style, typographic convention . . . . .                            | 34       |
| configuration                                                               |          |
| basic project settings . . . . .                                            | 54       |
| __low_level_init . . . . .                                                  | 135      |
| configuration symbols                                                       |          |
| for file input and output . . . . .                                         | 141      |
| for locale . . . . .                                                        | 142      |
| for printf and scanf. . . . .                                               | 139      |
| for strtod . . . . .                                                        | 146      |
| in library configuration files. . . . .                                     | 131, 136 |
| consistency, module . . . . .                                               | 154      |
| const                                                                       |          |
| declaring objects . . . . .                                                 | 286      |
| non-top level . . . . .                                                     | 193      |
| constants, placing in named segment . . . . .                               | 309      |
| __constrange(), symbol used in library. . . . .                             | 363      |
| __construction_by_bitwise_copy_allowed, symbol used<br>in library . . . . . | 363      |
| constseg (pragma directive) . . . . .                                       | 309      |
| const_cast (cast operator) . . . . .                                        | 196      |
| contents, of this guide . . . . .                                           | 30       |

|                                                        |          |
|--------------------------------------------------------|----------|
| control characters,                                    |          |
| implementation-defined behavior . . . . .              | 405      |
| conventions, used in this guide . . . . .              | 34       |
| __COP (intrinsic function) . . . . .                   | 334      |
| copyright notice . . . . .                             | 2        |
| __COP_get_registers (intrinsic function) . . . . .     | 335      |
| __COP_get_register32 (intrinsic function) . . . . .    | 335      |
| __COP_get_register64 (intrinsic function) . . . . .    | 335      |
| __COP_set_registers (intrinsic function) . . . . .     | 336      |
| __COP_set_register32 (intrinsic function) . . . . .    | 336      |
| __COP_set_register64 (intrinsic function) . . . . .    | 336      |
| __CORE__ (predefined symbol). . . . .                  | 348      |
| core                                                   |          |
| identifying . . . . .                                  | 348      |
| __core (runtime model attribute). . . . .              | 156      |
| __CORE_REVISION__ (predefined symbol) . . . . .        | 348      |
| cos (library function) . . . . .                       | 356      |
| cos (library routine) . . . . .                        | 146, 148 |
| cosf (library routine) . . . . .                       | 147–148  |
| cosl (library routine) . . . . .                       | 147–148  |
| __COUNTER__ (predefined symbol). . . . .               | 348      |
| __count_leading_zeros (intrinsic function). . . . .    | 336      |
| __count_trailing_zeros (intrinsic function). . . . .   | 337      |
| __cplusplus (predefined symbol) . . . . .              | 349      |
| --cpu (compiler option). . . . .                       | 247      |
| --cpu_info (compiler option). . . . .                  | 248      |
| CPU, specifying on command line for compiler . . . . . | 247      |
| csetjmp (DLIB header file) . . . . .                   | 360      |
| csignal (DLIB header file) . . . . .                   | 360      |
| cspy_support (pragma directive). . . . .               | 399, 413 |
| CSTACK (segment) . . . . .                             | 369      |
| example . . . . .                                      | 133      |
| <i>See also</i> stack                                  |          |
| cstartup (system startup code)                         |          |
| customizing system initialization. . . . .             | 135      |
| source files for (DLIB). . . . .                       | 132      |
| cstdarg (DLIB header file) . . . . .                   | 360      |
| cstdbool (DLIB header file) . . . . .                  | 360      |
| cstddef (DLIB header file) . . . . .                   | 360      |
| cstdio (DLIB header file) . . . . .                    | 360      |

- cstdlib (DLIB header file) . . . . . 360
  - cstring (DLIB header file) . . . . . 360
  - ctime (DLIB header file) . . . . . 360
  - ctype.h (library header file) . . . . . 357
  - cwctype.h (library header file) . . . . . 361
  - C\_INCLUDE (environment variable) . . . . . 232
  - C-SPY
    - debug support for C++ . . . . . 199
    - including debugging support . . . . . 126
    - interface to system termination . . . . . 135
    - Terminal I/O window, including debug support for . . . 127
  - C-STAT for static analysis, documentation for . . . . . 32
  - C++
    - See also Embedded C++ and Extended Embedded C++*
    - absolute location . . . . . 216–217
    - calling convention . . . . . 169
    - dynamic initialization in . . . . . 112
    - header files . . . . . 358
    - language extensions . . . . . 201
    - standard template library (STL) . . . . . 359
    - static member variables . . . . . 216–217
    - support for . . . . . 40
  - C++ header files . . . . . 359
  - C++ names, in assembler code . . . . . 170
  - C++ objects, placing in memory type . . . . . 66
  - C++ terminology . . . . . 34
  - C++-style comments . . . . . 187
  - C89
    - implementation-defined behavior . . . . . 407
    - support for . . . . . 187
  - c89 (compiler option) . . . . . 245
  - C99. *See* Standard C
- ## D
- D (compiler option) . . . . . 248
  - data
    - alignment of . . . . . 275
    - different ways of storing . . . . . 59
    - located, declaring extern . . . . . 216
    - placing . . . . . 214, 310, 365
      - at absolute location . . . . . 215
      - representation of . . . . . 275
      - storage . . . . . 59
      - verifying linked result . . . . . 117
  - data block (call frame information) . . . . . 182
  - data memory attributes, using . . . . . 63
  - data models . . . . . 66
    - configuration . . . . . 56
    - identifying (`__DATA_MODEL__`) . . . . . 349
    - large . . . . . 67
    - setting (`--data_model`) . . . . . 249
    - small . . . . . 67
  - data pointers . . . . . 281
  - data types . . . . . 276
    - avoiding signed . . . . . 211
    - floating point . . . . . 279
    - in C++ . . . . . 287
    - integer types . . . . . 276
  - dataseg (pragma directive) . . . . . 310
  - data\_alignment (pragma directive) . . . . . 309
  - `__DATA_MODEL__` (predefined symbol) . . . . . 349
  - `__data_model` (runtime model attribute) . . . . . 156
  - `--data_model` (compiler option) . . . . . 249
  - `__data`, symbol used in library . . . . . 362
  - `__data17` (extended keyword) . . . . . 295
  - data17 (memory type) . . . . . 61
  - `__data21` (extended keyword) . . . . . 295
  - data21 (memory type) . . . . . 61
  - `__data32` (extended keyword) . . . . . 296
  - data32 (memory type) . . . . . 61
  - `__DATE__` (predefined symbol) . . . . . 349
  - date (library function), configuring support for . . . . . 145
  - `__dbgreg` (extended keyword) . . . . . 296
  - DC32 (assembler directive) . . . . . 161
  - `--debug` (compiler option) . . . . . 249
  - debug information, including in object file . . . . . 249
  - decimal point, implementation-defined behavior . . . . . 405

|                                                              |          |
|--------------------------------------------------------------|----------|
| declarations                                                 |          |
| empty                                                        | 193      |
| in for loops                                                 | 187      |
| Kernighan & Ritchie                                          | 225      |
| of functions                                                 | 171      |
| declarations and statements, mixing                          | 187      |
| declarators, implementation-defined behavior in C89          | 412      |
| __DEFAULT_CODE_SEGMENT__ (predefined symbol)                 | 349      |
| __DEFAULT_CONST_SEGMENT__ (predefined symbol)                | 349      |
| __DEFAULT_DATA_SEGMENT__ (predefined symbol)                 | 349      |
| define_type_info (pragma directive)                          | 399, 413 |
| delete (keyword)                                             | 69       |
| denormalized numbers. <i>See</i> subnormal numbers           |          |
| --dependencies (compiler option)                             | 250      |
| deque (STL header file)                                      | 359      |
| destructors and interrupts, using                            | 199      |
| device description files, preconfigured for C-SPY            | 42       |
| DI (assembler instruction)                                   | 337      |
| diagnostic messages                                          | 235      |
| classifying as compilation errors                            | 251      |
| classifying as compilation remarks                           | 251      |
| classifying as compiler warnings                             | 252      |
| disabling compiler warnings                                  | 265      |
| disabling wrapping of in compiler                            | 265      |
| enabling compiler remarks                                    | 269      |
| listing all used by compiler                                 | 252      |
| suppressing in compiler                                      | 251      |
| --diagnostics_tables (compiler option)                       | 252      |
| diagnostics, implementation-defined behavior                 | 391      |
| diag_default (pragma directive)                              | 312      |
| --diag_error (compiler option)                               | 251      |
| diag_error (pragma directive)                                | 313      |
| --diag_remark (compiler option)                              | 251      |
| diag_remark (pragma directive)                               | 313      |
| --diag_suppress (compiler option)                            | 251      |
| diag_suppress (pragma directive)                             | 313      |
| --diag_warning (compiler option)                             | 252      |
| diag_warning (pragma directive)                              | 314      |
| DIFUNCT (segment)                                            | 112, 375 |
| directives                                                   |          |
| function for static overlay                                  | 178      |
| pragma                                                       | 42, 305  |
| directory, specifying as parameter                           | 238      |
| --disable_inline_asm_label_replacement (compiler option)     | 253      |
| __disable_interrupt (intrinsic function)                     | 337      |
| --discard_unused_publics (compiler option)                   | 253      |
| disclaimer                                                   | 2        |
| div_s (ETSI macro)                                           | 331      |
| DLIB                                                         | 357      |
| configurations                                               | 136      |
| configuring                                                  | 120, 253 |
| documentation                                                | 32       |
| including debug support                                      | 125      |
| naming convention                                            | 35       |
| reference information. <i>See</i> the online help system     | 355      |
| runtime environment                                          | 119      |
| --dlib_config (compiler option)                              | 253      |
| DLIB_Defaults.h (library configuration file)                 | 131, 136 |
| __DLIB_FILE_DESCRIPTOR (configuration symbol)                | 141      |
| document conventions                                         | 34       |
| documentation                                                |          |
| contents of this                                             | 30       |
| how to use this                                              | 29       |
| overview of guides                                           | 31       |
| who should read this                                         | 29       |
| domain errors, implementation-defined behavior               | 400      |
| domain errors, implementation-defined behavior in C89 (DLIB) | 414      |
| double (data type)                                           | 279      |
| avoiding                                                     | 211      |
| do_not_instantiate (pragma directive)                        | 399, 413 |
| dsp, enabling block of instructions                          | 56, 243  |
| dynamic initialization                                       | 131      |
| and C++                                                      | 112      |
| dynamic memory                                               | 69       |



- ## E
- e (compiler option) . . . . . 254
  - early\_initialization (pragma directive) . . . . . 399, 413
  - ec++ (compiler option) . . . . . 255
  - edition, of this guide . . . . . 2
  - eec++ (compiler option) . . . . . 255
  - EI (assembler instruction) . . . . . 337
  - Embedded C++ . . . . . 195
    - differences from C++ . . . . . 196
    - enabling . . . . . 255
    - function linkage . . . . . 171
    - language extensions . . . . . 195
    - overview . . . . . 195
  - Embedded C++ Technical Committee . . . . . 34
  - embedded systems, IAR special support for . . . . . 42
  - \_\_embedded\_cplusplus (predefined symbol) . . . . . 349
  - \_\_enable\_interrupt (intrinsic function) . . . . . 337
  - enable\_multibytes (compiler option) . . . . . 255
  - enable\_restrict (compiler option) . . . . . 256
  - enabling restrict keyword . . . . . 256
  - entry label, program . . . . . 132
  - enumerations
    - implementation-defined behavior . . . . . 397
    - implementation-defined behavior in C89 . . . . . 411
  - enums
    - data representation . . . . . 276
    - forward declarations of . . . . . 192
  - \_\_enum\_size (runtime model attribute) . . . . . 156
  - environment
    - implementation-defined behavior . . . . . 392
    - implementation-defined behavior in C89 . . . . . 407
    - runtime (DLIB) . . . . . 119
  - environment names, implementation-defined behavior . . . . . 393
  - environment variables
    - C\_INCLUDE . . . . . 232
    - QCCAVR32 . . . . . 232
  - environment (native),
    - implementation-defined behavior . . . . . 405
  - EQU (assembler directive) . . . . . 268
  - ERANGE . . . . . 400
  - ERANGE (C89) . . . . . 414
  - errno value at underflow,
    - implementation-defined behavior . . . . . 403
  - errno.h (library header file) . . . . . 357
  - error messages . . . . . 236
    - classifying for compiler . . . . . 251
  - error return codes . . . . . 234
  - error (pragma directive) . . . . . 314
  - error\_limit (compiler option) . . . . . 256
  - escape sequences, implementation-defined behavior . . . . . 393
  - EVBYTES1 (segment) . . . . . 376
  - EVBYTES2 (segment) . . . . . 376
  - EVBYTES3 (segment) . . . . . 376
  - EVSEG (segment) . . . . . 377
  - EVTAB (segment) . . . . . 377
  - EV100 (segment) . . . . . 377
  - \_\_exception (extended keyword) . . . . . 296
  - exception handlers . . . . . 75, 111
  - exception handling, missing from Embedded C++ . . . . . 196
  - exception table, start address for . . . . . 73
  - exception vectors . . . . . 112
  - exception (pragma directive) . . . . . 314
  - \_\_exchange\_memory (intrinsic function) . . . . . 337
  - exclude (stack usage control directive) . . . . . 384
  - execution modes . . . . . 303
  - \_Exit (library function) . . . . . 134
  - exit (library function) . . . . . 134
    - implementation-defined behavior . . . . . 403
    - implementation-defined behavior in C89 . . . . . 416
  - \_exit (library function) . . . . . 134
  - \_\_exit (library function) . . . . . 134
  - exp (library routine) . . . . . 146
  - expf (library routine) . . . . . 147
  - expl (library routine) . . . . . 147
  - export keyword, missing from Extended EC++ . . . . . 199
  - extended command line file
    - for compiler . . . . . 256
    - passing options . . . . . 232

|                                 |         |
|---------------------------------|---------|
| Extended Embedded C++           | 196     |
| enabling                        | 255     |
| extended keywords               | 289     |
| enabling (-e)                   | 254     |
| overview                        | 42      |
| summary                         | 292     |
| syntax                          |         |
| object attributes               | 292     |
| type attributes on data objects | 63, 290 |
| type attributes on functions    | 291     |
| extern "C" linkage              | 198     |
| extract_h (ETSI macro)          | 331     |
| extract_l (ETSI macro)          | 331     |

## F

|                                                                    |     |
|--------------------------------------------------------------------|-----|
| -f (compiler option)                                               | 256 |
| __far (extended keyword)                                           | 281 |
| __farfunc (function pointer)                                       | 281 |
| fatal error messages                                               | 236 |
| fdopen, in stdio.h                                                 | 361 |
| fegetrapdisable                                                    | 361 |
| fegetrapenable                                                     | 361 |
| FENV_ACCESS, implementation-defined behavior                       | 396 |
| fenv.h (library header file)                                       | 357 |
| additional C functionality                                         | 361 |
| fgetpos (library function), implementation-defined behavior        | 403 |
| fgetpos (library function), implementation-defined behavior in C89 | 416 |
| __FILE__ (predefined symbol)                                       | 350 |
| file buffering, implementation-defined behavior                    | 401 |
| file dependencies, tracking                                        | 250 |
| file paths, specifying for #include files                          | 258 |
| file position, implementation-defined behavior                     | 401 |
| file streams lock interface                                        | 151 |
| file (zero-length), implementation-defined behavior                | 401 |
| filename                                                           |     |
| extension for device description files                             | 42  |

|                                                                 |               |
|-----------------------------------------------------------------|---------------|
| extension for header files                                      | 41            |
| of object file                                                  | 267           |
| search procedure for                                            | 232           |
| specifying as parameter                                         | 238           |
| filenames (legal), implementation-defined behavior              | 401           |
| fileno, in stdio.h                                              | 361           |
| files, implementation-defined behavior                          |               |
| handling of temporary                                           | 402           |
| multibyte characters in                                         | 402           |
| opening                                                         | 402           |
| FlashVault                                                      |               |
| using to implement middleware                                   | 84            |
| __flashvault (extended keyword)                                 | 297           |
| __flashvault_impl (extended keyword)                            | 297           |
| flashvault_vector (pragma directive)                            | 315           |
| flashvault, enabling block of instructions                      | 56            |
| float (data type)                                               | 279           |
| floating-point constants                                        |               |
| hexadecimal notation                                            | 187           |
| hints                                                           | 212           |
| floating-point environment, accessing or not                    | 327           |
| floating-point expressions                                      |               |
| contracting or not                                              | 327           |
| floating-point format                                           | 279           |
| hints                                                           | 211           |
| implementation-defined behavior                                 | 395           |
| implementation-defined behavior in C89                          | 410           |
| special cases                                                   | 280           |
| 32-bits                                                         | 280           |
| 64-bits                                                         | 280           |
| floating-point status flags                                     | 361           |
| float.h (library header file)                                   | 357           |
| FLT_EVAL_METHOD, implementation-defined behavior                | 395, 400, 404 |
| FLT_ROUNDS, implementation-defined behavior                     | 395, 404      |
| fmod (library function), implementation-defined behavior in C89 | 414           |
| for loops, declarations in                                      | 187           |

- formats
    - floating-point values . . . . . 279
    - standard IEEE (floating point) . . . . . 279
  - fpu, enabling block of instructions . . . . . 56, 244
  - FP\_CONTRACT, implementation-defined behavior . . . . . 396
  - fp\_implementation (compiler option) . . . . . 257
  - fragmentation, of heap memory . . . . . 69
  - free (library function). *See also* heap . . . . . 69
  - fsetpos (library function), implementation-defined behavior . . . . . 403
  - fstream (library header file) . . . . . 359
  - ftell (library function), implementation-defined behavior . 403
    - in C89 . . . . . 416
  - Full DLIB (library configuration) . . . . . 136
  - \_\_func\_\_ (predefined symbol) . . . . . 194, 350
  - FUNCALL (assembler directive) . . . . . 178
  - \_\_FUNCTION\_\_ (predefined symbol) . . . . . 194, 350
  - function calls
    - calling convention . . . . . 170
    - eliminating overhead of by inlining . . . . . 87
    - large code model . . . . . 179
    - preserved registers across. . . . . 172
    - small code model . . . . . 179
  - function declarations, Kernighan & Ritchie . . . . . 225
  - function directives for static overlay . . . . . 178
  - function execution, in RAM . . . . . 83
  - function inlining (compiler transformation) . . . . . 222
    - disabling (--no\_inline) . . . . . 262
  - function pointers . . . . . 281
  - function prototypes . . . . . 224
    - enforcing . . . . . 270
  - function return addresses . . . . . 176
  - function type information, omitting in object output. . . . 266
  - FUNCTION (assembler directive) . . . . . 178
  - function (pragma directive) . . . . . 399, 413
  - function (stack usage control directive). . . . . 385
  - functional (STL header file) . . . . . 359
  - functions . . . . . 71
    - acall . . . . . 77
    - calling in different code models . . . . . 178
    - declaring . . . . . 171, 224
    - inlining. . . . . 187, 222, 224, 317
    - interrupt . . . . . 73, 79
    - intrinsic . . . . . 159, 224
    - monitor . . . . . 79
    - omitting type info . . . . . 266
    - parameters . . . . . 173
    - placing in memory . . . . . 214, 217
    - placing segments for . . . . . 108, 111
    - recursive
      - avoiding . . . . . 224
      - storing data on stack . . . . . 68
    - reentrancy (DLIB) . . . . . 356
    - related extensions. . . . . 71
    - return values from . . . . . 175
    - scall . . . . . 78
    - special function types. . . . . 73
    - verifying linked result . . . . . 117
  - function\_effects (pragma directive). . . . . 399, 413
  - function-spec (in stack usage control file). . . . . 387
  - FVVEC (segment) . . . . . 378
- ## G
- getenv (library function), configuring support for . . . . . 144
  - getw, in stdio.h . . . . . 361
  - getzone (library function), configuring support for . . . . . 145
  - getzone.c . . . . . 145
  - \_\_get\_system\_register (intrinsic function) . . . . . 338
  - \_\_get\_interrupt\_state (intrinsic function) . . . . . 338
  - \_\_get\_system\_register (intrinsic function) . . . . . 339
  - \_\_get\_user\_context (intrinsic function). . . . . 339
  - global arrays, accessing . . . . . 180
  - global variables
    - accessing . . . . . 180
    - affected by static clustering . . . . . 223
    - handled during system termination . . . . . 134
    - hints for not using . . . . . 224
    - initialized during system startup . . . . . 133

--guard\_calls (compiler option) . . . . . 257  
 guidelines, reading . . . . . 29

## H

handler (pragma directive) . . . . . 74, 315  
 Harbison, Samuel P. . . . . 33  
 hardware support in compiler . . . . . 119  
 hash\_map (STL header file) . . . . . 359  
 hash\_set (STL header file) . . . . . 359  
 \_\_has\_constructor, symbol used in library . . . . . 363  
 \_\_has\_destructor, symbol used in library . . . . . 363  
 \_\_HAS\_DSP\_INSTRUCTIONS\_\_ (predefined symbol) . 350  
 \_\_HAS\_FPU\_INSTRUCTIONS\_\_ (predefined symbol) . 350  
 \_\_HAS\_RMW\_INSTRUCTIONS\_\_ (predefined symbol) 351  
 \_\_HAS\_SIMD\_INSTRUCTIONS\_\_ (predefined symbol) 351  
 hdrstop (pragma directive) . . . . . 399, 413  
 header files  
   C . . . . . 357  
   C++ . . . . . 358–359  
   library . . . . . 355  
   special function registers . . . . . 227  
   STL . . . . . 359  
   DLib\_Defaults.h . . . . . 131, 136  
   including stdbool.h for bool . . . . . 276  
   including stddef.h for wchar\_t . . . . . 277  
 header names, implementation-defined behavior . . . . . 397  
 --header\_context (compiler option) . . . . . 257  
 heap  
   dynamic memory . . . . . 69  
   segments for . . . . . 109  
   storing data . . . . . 60  
   VLA allocated on . . . . . 272  
 heap segments  
   DLIB . . . . . 206  
   placing . . . . . 110  
 heap size  
   and standard I/O . . . . . 206  
   changing default . . . . . 109–110

HEAP (section) . . . . . 206  
 heap (zero-sized), implementation-defined behavior . . . . . 403  
 hints  
   for good code generation . . . . . 223  
   implementation-defined behavior . . . . . 396  
   using efficient data types . . . . . 211  
 HTAB (segment) . . . . . 378

## I

-I (compiler option) . . . . . 258  
 IAR Command Line Build Utility . . . . . 130  
 IAR Systems Technical Support . . . . . 236  
 iarbuild.exe (utility) . . . . . 130  
 \_\_iar\_cos\_accurate (library routine) . . . . . 148  
 \_\_iar\_cos\_accuratef (library routine) . . . . . 148  
 \_\_iar\_cos\_accuratef (library function) . . . . . 356  
 \_\_iar\_cos\_accuratel (library routine) . . . . . 148  
 \_\_iar\_cos\_accuratel (library function) . . . . . 356  
 \_\_iar\_cos\_small (library routine) . . . . . 146  
 \_\_iar\_cos\_smallf (library routine) . . . . . 147  
 \_\_iar\_cos\_smallll (library routine) . . . . . 147  
 \_\_IAR\_DLIB\_PERTHREAD\_INIT\_SIZE (macro) . . . . . 152  
 \_\_IAR\_DLIB\_PERTHREAD\_SIZE (macro) . . . . . 152  
 \_\_IAR\_DLIB\_PERTHREAD\_SYMBOL\_OFFSET  
 (symbolptr) . . . . . 152  
 \_\_iar\_exp\_small (library routine) . . . . . 146  
 \_\_iar\_exp\_smallf (library routine) . . . . . 147  
 \_\_iar\_exp\_smallll (library routine) . . . . . 147  
 \_\_iar\_log\_small (library routine) . . . . . 146  
 \_\_iar\_log\_smallf (library routine) . . . . . 147  
 \_\_iar\_log\_smallll (library routine) . . . . . 147  
 \_\_iar\_log10\_small (library routine) . . . . . 146  
 \_\_iar\_log10\_smallf (library routine) . . . . . 147  
 \_\_iar\_log10\_smallll (library routine) . . . . . 147  
 \_\_iar\_Powf (library routine) . . . . . 148  
 \_\_iar\_Powl (library routine) . . . . . 148  
 \_\_iar\_Pow\_accurate (library routine) . . . . . 148  
 \_\_iar\_pow\_accurate (library routine) . . . . . 148

- `__iar_Pow_accuratef` (library routine) . . . . . 148
- `__iar_pow_accuratef` (library routine). . . . . 148
- `__iar_pow_accuratef` (library function). . . . . 356
- `__iar_Pow_accuratel` (library routine). . . . . 148
- `__iar_pow_accuratel` (library routine). . . . . 148
- `__iar_pow_accuratel` (library function). . . . . 356
- `__iar_pow_small` (library routine). . . . . 146
- `__iar_pow_smallf` (library routine) . . . . . 147
- `__iar_pow_smallll` (library routine) . . . . . 147
- `__iar_program_start` (label). . . . . 132
- `__iar_Sin` (library routine) . . . . . 146
- `__iar_Sinf` (library routine). . . . . 148
- `__iar_Sinl` (library routine). . . . . 148
- `__iar_Sin_accurate` (library routine) . . . . . 148
- `__iar_sin_accurate` (library routine) . . . . . 148
- `__iar_Sin_accuratef` (library routine) . . . . . 148
- `__iar_sin_accuratef` (library routine). . . . . 148
- `__iar_sin_accuratef` (library function). . . . . 356
- `__iar_Sin_accuratel` (library routine) . . . . . 148
- `__iar_sin_accuratel` (library routine). . . . . 148
- `__iar_sin_accuratel` (library function). . . . . 356
- `__iar_Sin_small` (library routine) . . . . . 146
- `__iar_sin_small` (library routine). . . . . 146
- `__iar_Sin_smallf` (library routine). . . . . 147
- `__iar_sin_smallf` (library routine) . . . . . 147
- `__iar_Sin_smallll` (library routine). . . . . 147
- `__iar_sin_smallll` (library routine) . . . . . 147
- `__IAR_SYSTEMS_ICC__` (predefined symbol) . . . . . 351
- `__iar_tan_accurate` (library routine) . . . . . 148
- `__iar_tan_accuratef` (library routine). . . . . 148
- `__iar_tan_accuratef` (library function). . . . . 356
- `__iar_tan_accuratel` (library routine). . . . . 148
- `__iar_tan_accuratel` (library function). . . . . 356
- `__iar_tan_small` (library routine) . . . . . 146
- `__iar_tan_smallf` (library routine). . . . . 147
- `__iar_tan_smallll` (library routine) . . . . . 147
- `__ICCAVR32__` (predefined symbol). . . . . 351
- icons, in this guide . . . . . 34
- IDE
  - building a library from . . . . . 130
  - overview of build tools. . . . . 39
- identifiers, implementation-defined behavior . . . . . 393
- identifiers, implementation-defined behavior in C89 . . . . 408
- IEEE format, floating-point values . . . . . 279
- `__important_typedef` (pragma directive). . . . . 399, 413
- `__imported` (extended keyword) . . . . . 298
- include files
  - including before source files . . . . . 268
  - specifying . . . . . 232
- `include_alias` (pragma directive) . . . . . 316
- infinity . . . . . 280
- infinity (style for printing), implementation-defined behavior . . . . . 402
- inheritance, in Embedded C++ . . . . . 195
- initialization
  - dynamic . . . . . 131
  - single-value . . . . . 193
- initializers, static . . . . . 192
- INITTAB (segment) . . . . . 379
- inline assembler . . . . . 160
  - avoiding . . . . . 224
  - for passing values between C and assembler . . . . . 228
  - See also* assembler language interface
- inline functions . . . . . 187
  - in compiler. . . . . 222
- inline (pragma directive). . . . . 317
- inlining functions . . . . . 87
  - implementation-defined behavior. . . . . 396
- installation directory . . . . . 34
- instantiate (pragma directive) . . . . . 399, 413
- instruction scheduling (compiler option). . . . . 223
- int (data type) signed and unsigned. . . . . 276
- integer types . . . . . 276
  - casting . . . . . 282
  - implementation-defined behavior. . . . . 394
- `intptr_t` . . . . . 282
- `ptrdiff_t` . . . . . 282
- `size_t` . . . . . 282

|                                                         |         |
|---------------------------------------------------------|---------|
| uintptr_t                                               | 282     |
| integers, implementation-defined behavior in C89        | 409     |
| integral promotion                                      | 225     |
| internal error                                          | 236     |
| __interrupt (extended keyword)                          | 74, 298 |
| interrupt controller initialization, __init_ihandler    | 132     |
| interrupt functions                                     | 73      |
| placement in memory                                     | 112     |
| interrupt handler. <i>See</i> interrupt service routine |         |
| interrupt service routine                               | 73      |
| interrupt state, restoring                              | 341     |
| interrupt vector                                        | 74      |
| specifying with pragma directive                        | 328     |
| interrupt vector table                                  |         |
| in linker configuration file                            | 112     |
| interrupts                                              |         |
| disabling                                               | 298     |
| during function execution                               | 79      |
| processor state                                         | 68      |
| using with EC++ destructors                             | 199     |
| intptr_t (integer type)                                 | 282     |
| __intrinsic (extended keyword)                          | 298     |
| intrinsic functions                                     | 224     |
| overview                                                | 159     |
| summary                                                 | 329     |
| intrinsics.h (header file)                              | 329     |
| inttypes.h (library header file)                        | 358     |
| INTVEC (segment)                                        | 112     |
| invocation syntax                                       | 231     |
| iomaniop (library header file)                          | 359     |
| ios (library header file)                               | 359     |
| iosfwd (library header file)                            | 359     |
| iostream (library header file)                          | 359     |
| iso646.h (library header file)                          | 358     |
| istream (library header file)                           | 359     |
| italic style, in this guide                             | 34      |
| iterator (STL header file)                              | 359     |
| I/O register. <i>See</i> SFR                            |         |

## J

|                      |    |
|----------------------|----|
| Josuttis, Nicolai M. | 33 |
|----------------------|----|

## K

|                                           |          |
|-------------------------------------------|----------|
| keep_definition (pragma directive)        | 399, 413 |
| Kernighan & Ritchie function declarations | 225      |
| disallowing                               | 270      |
| Kernighan, Brian W.                       | 33       |
| keywords                                  | 289      |
| extended, overview of                     | 42       |

## L

|                             |          |
|-----------------------------|----------|
| -l (compiler option)        | 258      |
| for creating skeleton code  | 168      |
| labels                      | 193      |
| assembler, making public    | 268      |
| __iar_program_start         | 132      |
| __program_start             | 132      |
| Labrosse, Jean J.           | 33       |
| Lajoie, Josée               | 33       |
| language extensions         |          |
| Embedded C++                | 195      |
| enabling using pragma       | 317      |
| enabling (-e)               | 254      |
| language overview           | 40       |
| language (pragma directive) | 317      |
| Large (code model)          |          |
| function calls              | 179      |
| libraries                   |          |
| reason for using            | 48       |
| standard template library   | 359      |
| using a prebuilt            | 121      |
| library configuration files |          |
| DLIB                        | 136      |
| DLib_Defaults.h             | 131, 136 |
| modifying                   | 131      |

- specifying . . . . . 253
- library documentation . . . . . 355
- library features, missing from Embedded C++ . . . . . 196
- library functions
  - summary, DLIB . . . . . 357
  - online help for . . . . . 32
- library header files . . . . . 355
- library modules
  - creating . . . . . 259
  - overriding . . . . . 129
- library object files . . . . . 355
- library options, setting . . . . . 58
- library project, building using a template . . . . . 130
- library\_default\_requirements (pragma directive) . . . 399, 413
- library\_module (compiler option) . . . . . 259
- library\_provides (pragma directive) . . . . . 399, 413
- library\_requirement\_override (pragma directive) . . . 399, 413
- lightbulb icon, in this guide . . . . . 35
- limits.h (library header file) . . . . . 358
- \_\_LINE\_\_ (predefined symbol) . . . . . 351
- linkage, C and C++ . . . . . 171
- linker. . . . . 89
- linker configuration file . . . . . 92
  - for placing code and data . . . . . 92
  - in depth . . . . . 383
  - overview of . . . . . 383
  - using the -P command . . . . . 107
  - using the -Z command . . . . . 107
- linker map file. . . . . 118
- linker options
  - typographic convention . . . . . 34
- linker segment. *See* segment
- linking
  - from the command line . . . . . 54
  - in the build process . . . . . 48
  - introduction . . . . . 89
  - process for . . . . . 91
- Lippman, Stanley B. . . . . 33
- list (STL header file). . . . . 359
- listing, generating . . . . . 258
- literals, compound. . . . . 187
- literature, recommended . . . . . 33
- local variables, *See* auto variables
- locale
  - adding support for in library . . . . . 143
  - changing at runtime . . . . . 143
  - implementation-defined behavior. . . . . 394, 404
  - removing support for . . . . . 143
  - support for . . . . . 142
- locale.h (library header file) . . . . . 358
- located data segments . . . . . 108
- located data, declaring extern . . . . . 216
- location (pragma directive) . . . . . 215, 318
- LOCFRAME (assembler directive). . . . . 178
- log (library routine). . . . . 146
- logf (library routine) . . . . . 147
- logl (library routine) . . . . . 147
- log10 (library routine). . . . . 146
- log10f (library routine) . . . . . 147
- log10l (library routine) . . . . . 147
- long double (data type). . . . . 279
- long float (data type), synonym for double . . . . . 192
- long long (data type)
  - avoiding . . . . . 211
- long long (data type) signed and unsigned . . . . . 276
- long (data type) signed and unsigned . . . . . 276
- longjmp, restrictions for using . . . . . 357
- loop unrolling (compiler transformation) . . . . . 221
  - disabling . . . . . 264
- loop-invariant expressions. . . . . 222
- \_\_low\_level\_init . . . . . 132
  - customizing . . . . . 135
  - initialization phase . . . . . 50
- low\_level\_init.c. . . . . 132
- low\_level\_init.s99. . . . . 132
- low-level processor operations . . . . . 188, 329
  - accessing . . . . . 159
- \_\_lseek (library function) . . . . . 141

|                                    |     |
|------------------------------------|-----|
| L_abs (ETSI macro) . . . . .       | 332 |
| L_add (ETSI macro) . . . . .       | 332 |
| L_deposit_c (ETSI macro) . . . . . | 332 |
| L_deposit_l (ETSI macro) . . . . . | 332 |
| L_mac (ETSI macro) . . . . .       | 332 |
| L_macNs (ETSI macro) . . . . .     | 332 |
| L_msu (ETSI macro) . . . . .       | 332 |
| L_msuNs (ETSI macro) . . . . .     | 332 |
| L_mult (ETSI macro) . . . . .      | 332 |
| L_negate (ETSI macro) . . . . .    | 332 |
| L_sat (ETSI macro) . . . . .       | 332 |
| L_shl (ETSI macro) . . . . .       | 332 |
| L_shr (ETSI macro) . . . . .       | 332 |
| L_shr_r (ETSI macro) . . . . .     | 332 |
| L_sub (ETSI macro) . . . . .       | 332 |
| L_sub_c (ETSI macro) . . . . .     | 332 |

## M

### macros

|                                                              |          |
|--------------------------------------------------------------|----------|
| embedded in #pragma optimize . . . . .                       | 321      |
| ERANGE (in errno.h) . . . . .                                | 400, 414 |
| inclusion of assert . . . . .                                | 353      |
| NULL, implementation-defined behavior . . . . .              | 401      |
| in C89 for DLIB . . . . .                                    | 414      |
| substituted in #pragma directives . . . . .                  | 188      |
| variadic . . . . .                                           | 187      |
| --macro_positions_in_diagnostics (compiler option) . . . . . | 259      |
| mac_r (ETSI macro) . . . . .                                 | 332      |
| main (function)                                              |          |
| definition (C89) . . . . .                                   | 407      |
| implementation-defined behavior . . . . .                    | 392      |
| malloc (library function)                                    |          |
| <i>See also</i> heap . . . . .                               | 69       |
| implementation-defined behavior in C89 . . . . .             | 416      |
| Mann, Bernhard . . . . .                                     | 33       |
| map (STL header file) . . . . .                              | 359      |
| map, linker . . . . .                                        | 118      |

|                                                               |          |
|---------------------------------------------------------------|----------|
| math functions rounding mode,                                 |          |
| implementation-defined behavior . . . . .                     | 404      |
| math functions (library functions) . . . . .                  | 146      |
| math.h (library header file) . . . . .                        | 358      |
| __max (intrinsic function) . . . . .                          | 339      |
| max recursion depth (stack usage control directive) . . . . . | 385      |
| MB_LEN_MAX, implementation-defined behavior . . . . .         | 404      |
| memory                                                        |          |
| accessing . . . . .                                           | 60, 180  |
| using data20 method . . . . .                                 | 181      |
| allocating in C++ . . . . .                                   | 69       |
| dynamic . . . . .                                             | 69       |
| heap . . . . .                                                | 69       |
| non-initialized . . . . .                                     | 228      |
| RAM, saving . . . . .                                         | 224      |
| releasing in C++ . . . . .                                    | 69       |
| stack . . . . .                                               | 68       |
| saving . . . . .                                              | 224      |
| used by global or static variables . . . . .                  | 60       |
| memory access methods                                         |          |
| data20 . . . . .                                              | 181      |
| data32 . . . . .                                              | 181      |
| system register . . . . .                                     | 181      |
| memory clobber . . . . .                                      | 161      |
| memory management, type-safe . . . . .                        | 195      |
| memory map                                                    |          |
| initializing SFRs . . . . .                                   | 135      |
| linker configuration for . . . . .                            | 105      |
| memory placement                                              |          |
| of linker segments . . . . .                                  | 92       |
| using type definitions . . . . .                              | 65       |
| memory segment. <i>See</i> segment                            |          |
| memory types . . . . .                                        | 60       |
| C++ . . . . .                                                 | 66       |
| placing variables in . . . . .                                | 66       |
| specifying . . . . .                                          | 63       |
| structures . . . . .                                          | 65       |
| summary . . . . .                                             | 63       |
| memory (pragma directive) . . . . .                           | 399, 413 |
| memory (STL header file) . . . . .                            | 359      |



\_\_memory\_of  
     symbol used in library . . . . . 363  
 message (pragma directive) . . . . . 319  
 messages  
     disabling . . . . . 270  
     forcing . . . . . 319  
 Meyers, Scott . . . . . 33  
 --mfc (compiler option) . . . . . 260  
 migration  
     from earlier IAR compilers . . . . . 32  
 \_\_min (intrinsic function) . . . . . 339  
 --minimize\_constant\_tables (compiler option) . . . . . 260  
 MISRA C, documentation . . . . . 32  
 --misrac\_verbose (compiler option) . . . . . 241  
 --misrac1998 (compiler option) . . . . . 241  
 --misrac2004 (compiler option) . . . . . 241  
 MMU, memory management unit . . . . . 114  
 mode changing, implementation-defined behavior . . . . . 402  
 modes . . . . . 303  
 module consistency . . . . . 154  
     rtmodel . . . . . 323  
 module map, in linker map file . . . . . 118  
 module name, specifying (--module\_name) . . . . . 260  
 module summary, in linker map file . . . . . 118  
 --module\_name (compiler option) . . . . . 260  
 module\_name (pragma directive) . . . . . 399, 413  
 module-spec (in stack usage control file) . . . . . 387  
 \_\_monitor (extended keyword) . . . . . 298  
 monitor functions . . . . . 79, 298  
 msu\_r (ETSI macro) . . . . . 332  
 mult (ETSI macro) . . . . . 332  
 multibyte character support . . . . . 255  
 multibyte characters, implementation-defined  
 behavior . . . . . 393, 405  
 multiple inheritance  
     in Extended EC++ . . . . . 196  
     missing from Embedded C++ . . . . . 196  
 multithreaded environment . . . . . 149  
 multi-file compilation . . . . . 219  
 mult\_r (ETSI macro) . . . . . 333

mutable attribute, in Extended EC++ . . . . . 196, 200

## N

name (in stack usage control file) . . . . . 388  
 names block (call frame information) . . . . . 182  
 namespace support  
     in Extended EC++ . . . . . 196, 200  
     missing from Embedded C++ . . . . . 196  
 naming conventions . . . . . 35  
 NaN  
     implementation of . . . . . 281  
     implementation-defined behavior . . . . . 402  
 native environment,  
 implementation-defined behavior . . . . . 405  
 NDEBUG (preprocessor symbol) . . . . . 353  
 \_\_near (extended keyword) . . . . . 281  
 \_\_nearfunc (function pointer) . . . . . 281  
 negate (ETSI macro) . . . . . 333  
 \_\_nested (extended keyword) . . . . . 299  
 nesting interrupts . . . . . 74  
 new (keyword) . . . . . 69  
 new (library header file) . . . . . 359  
 no calls from (stack usage control directive) . . . . . 386  
 non-initialized variables, hints for . . . . . 228  
 non-scalar parameters, avoiding . . . . . 224  
 NOP (assembler instruction) . . . . . 340  
 \_\_noreturn (extended keyword) . . . . . 301  
 Normal DLIB (library configuration) . . . . . 136  
 norm\_l (ETSI macro) . . . . . 333  
 norm\_s (ETSI macro) . . . . . 333  
 Not a number (NaN) . . . . . 281  
 \_\_no\_alloc (extended keyword) . . . . . 299  
 \_\_no\_alloc\_str (operator) . . . . . 300  
 \_\_no\_alloc\_str16 (operator) . . . . . 300  
 \_\_no\_alloc16 (extended keyword) . . . . . 299  
 --no\_clustering (compiler option) . . . . . 261  
 --no\_code\_motion (compiler option) . . . . . 261  
 --no\_cse (compiler option) . . . . . 261

|                                                                       |          |
|-----------------------------------------------------------------------|----------|
| <code>__no_init</code> (extended keyword) . . . . .                   | 228, 301 |
| <code>--no_inline</code> (compiler option) . . . . .                  | 262      |
| <code>__no_operation</code> (intrinsic function) . . . . .            | 340      |
| <code>--no_path_in_file_macros</code> (compiler option) . . . . .     | 262      |
| <code>no_pch</code> (pragma directive) . . . . .                      | 399, 413 |
| <code>--no_scheduling</code> (compiler option) . . . . .              | 262      |
| <code>--no_size_constraints</code> (compiler option) . . . . .        | 263      |
| <code>--no_static_destruction</code> (compiler option) . . . . .      | 263      |
| <code>--no_system_include</code> (compiler option) . . . . .          | 263      |
| <code>--no_tbaa</code> (compiler option) . . . . .                    | 264      |
| <code>--no_typedefs_in_diagnostics</code> (compiler option) . . . . . | 264      |
| <code>--no_unroll</code> (compiler option) . . . . .                  | 264      |
| <code>--no_warnings</code> (compiler option) . . . . .                | 265      |
| <code>--no_wrap_diagnostics</code> (compiler option) . . . . .        | 265      |
| NULL                                                                  |          |
| implementation-defined behavior . . . . .                             | 401      |
| implementation-defined behavior in C89 (DLIB) . . . . .               | 414      |
| pointer constant, relaxation to Standard C . . . . .                  | 192      |
| numeric conversion functions,                                         |          |
| implementation-defined behavior . . . . .                             | 405      |
| numeric (STL header file) . . . . .                                   | 359      |

## O

|                                                                         |          |
|-------------------------------------------------------------------------|----------|
| <code>-O</code> (compiler option) . . . . .                             | 265      |
| <code>-o</code> (compiler option) . . . . .                             | 266      |
| object attributes . . . . .                                             | 292      |
| object filename, specifying ( <code>-o</code> ) . . . . .               | 267      |
| object module name, specifying ( <code>--module_name</code> ) . . . . . | 260      |
| <code>object_attribute</code> (pragma directive) . . . . .              | 228, 319 |
| <code>--omit_types</code> (compiler option) . . . . .                   | 266      |
| <code>once</code> (pragma directive) . . . . .                          | 399, 413 |
| <code>--only_stdout</code> (compiler option) . . . . .                  | 266      |
| <code>__open</code> (library function) . . . . .                        | 141      |
| operators                                                               |          |
| <i>See also</i> <code>@</code> (operator)                               |          |
| for cast                                                                |          |
| in Extended EC++. . . . .                                               | 196      |
| missing from Embedded C++. . . . .                                      | 196      |

|                                                                        |         |
|------------------------------------------------------------------------|---------|
| for segment control . . . . .                                          | 191     |
| precision for 32-bit float . . . . .                                   | 280     |
| precision for 64-bit float . . . . .                                   | 280     |
| sizeof, implementation-defined behavior . . . . .                      | 404     |
| variants for cast . . . . .                                            | 200     |
| <code>_Pragma</code> (preprocessor) . . . . .                          | 187     |
| <code>__ALIGNOF__</code> , for alignment control . . . . .             | 190     |
| ?, language extensions for . . . . .                                   | 201     |
| optimization                                                           |         |
| clustering, disabling . . . . .                                        | 261     |
| code motion, disabling . . . . .                                       | 261     |
| common sub-expression elimination, disabling . . . . .                 | 261     |
| configuration . . . . .                                                | 57      |
| disabling . . . . .                                                    | 221     |
| function inlining, disabling ( <code>--no_inline</code> ) . . . . .    | 262     |
| hints . . . . .                                                        | 223     |
| loop unrolling, disabling . . . . .                                    | 264     |
| scheduling, disabling . . . . .                                        | 262     |
| specifying ( <code>-O</code> ) . . . . .                               | 265     |
| techniques . . . . .                                                   | 221     |
| type-based alias analysis, disabling ( <code>--tbaa</code> ) . . . . . | 264     |
| using inline assembler code . . . . .                                  | 161     |
| using pragma directive . . . . .                                       | 320     |
| optimization levels . . . . .                                          | 220     |
| optimize (pragma directive) . . . . .                                  | 320     |
| option parameters . . . . .                                            | 237     |
| options, compiler. <i>See</i> compiler options                         |         |
| Oram, Andy . . . . .                                                   | 33      |
| ostream (library header file) . . . . .                                | 359     |
| output                                                                 |         |
| from preprocessor . . . . .                                            | 268     |
| specifying for linker . . . . .                                        | 54      |
| <code>--output</code> (compiler option) . . . . .                      | 266     |
| overhead, reducing . . . . .                                           | 221–222 |

## P

|                                                    |          |
|----------------------------------------------------|----------|
| pack (pragma directive) . . . . .                  | 283, 321 |
| <code>__packed</code> (extended keyword) . . . . . | 301      |

- packed structure types . . . . . 283
- parameters
  - function . . . . . 173
  - hidden . . . . . 173
  - non-scalar, avoiding . . . . . 224
  - register . . . . . 173–174
  - rules for specifying a file or directory . . . . . 238
  - specifying . . . . . 239
  - stack . . . . . 173–174
  - typographic convention . . . . . 34
- \_\_PART\_\_ (predefined symbol) . . . . . 351
- part number, of this guide . . . . . 2
- pending\_instantiations (compiler option) . . . . . 267
- permanent registers . . . . . 172
- perorr (library function),  
implementation-defined behavior in C89 . . . . . 416
- placement
  - code and data . . . . . 365
  - in named segments . . . . . 217
  - of code and data, introduction to . . . . . 92
- plain char, implementation-defined behavior . . . . . 393
- pointer types . . . . . 281
  - mixing . . . . . 192
- pointers
  - casting . . . . . 282
  - data . . . . . 281
  - function . . . . . 281
  - implementation-defined behavior . . . . . 396
  - implementation-defined behavior in C89 . . . . . 410
- polymorphism, in Embedded C++ . . . . . 195
- porting, code containing pragma directives . . . . . 307
- possible calls (stack usage control directive) . . . . . 386
- pow (library routine) . . . . . 146, 148
  - alternative implementation of . . . . . 356
- powf (library routine) . . . . . 147–148
- powl (library routine) . . . . . 147–148
- pragma directives . . . . . 42
  - summary . . . . . 305
  - exception . . . . . 314
  - for absolute located data . . . . . 215
  - handler . . . . . 315
  - list of all recognized . . . . . 398
  - list of all recognized (C89) . . . . . 413
  - pack . . . . . 283, 321
  - shadow\_registers . . . . . 325
- \_Pragma (preprocessor operator) . . . . . 187
- predefined symbols
  - overview . . . . . 43
  - summary . . . . . 348
- predef\_macro (compiler option) . . . . . 267
- \_\_prefetch\_cache (intrinsic function) . . . . . 340
- preinclude (compiler option) . . . . . 268
- preprocess (compiler option) . . . . . 268
- preprocessor
  - operator (\_Pragma) . . . . . 187
  - output . . . . . 268
- preprocessor directives
  - comments at the end of . . . . . 193
  - implementation-defined behavior . . . . . 397
  - implementation-defined behavior in C89 . . . . . 412
  - #pragma . . . . . 305
- preprocessor extensions
  - \_\_VA\_ARGS\_\_ . . . . . 187
  - #warning message . . . . . 353
- preprocessor symbols . . . . . 348
  - defining . . . . . 248
- preserved registers . . . . . 172
- \_\_PRETTY\_FUNCTION\_\_ (predefined symbol) . . . . . 352
- primitives, for special functions . . . . . 73
- printf formatter, selecting . . . . . 124
- printf (library function) . . . . . 123
  - choosing formatter . . . . . 123
  - configuration symbols . . . . . 139
  - implementation-defined behavior . . . . . 402
  - implementation-defined behavior in C89 . . . . . 415
- \_\_printf\_args (pragma directive) . . . . . 322
- printing characters, implementation-defined behavior . . . 405
- processor configuration . . . . . 55

|                                                      |          |
|------------------------------------------------------|----------|
| processor operations                                 |          |
| accessing                                            | 159      |
| low-level                                            | 188, 329 |
| program entry label                                  | 132      |
| program termination, implementation-defined behavior | 392      |
| programming hints                                    | 223      |
| __program_start (label)                              | 132      |
| projects                                             |          |
| basic settings for                                   | 54       |
| setting up for a library                             | 130      |
| prototypes, enforcing                                | 270      |
| ptrdiff_t (integer type)                             | 282      |
| PUBLIC (assembler directive)                         | 268      |
| publication date, of this guide                      | 2        |
| --public_equ (compiler option)                       | 268      |
| public_equ (pragma directive)                        | 322      |
| putenv (library function), absent from DLIB          | 144      |
| putw, in stdio.h                                     | 361      |

## Q

|                                        |     |
|----------------------------------------|-----|
| QCCAVR32 (environment variable)        | 232 |
| qualifiers                             |     |
| const and volatile                     | 284 |
| implementation-defined behavior        | 397 |
| implementation-defined behavior in C89 | 411 |
| queue (STL header file)                | 360 |

## R

|                                                   |     |
|---------------------------------------------------|-----|
| -r (compiler option)                              | 249 |
| raise (library function), configuring support for | 145 |
| raise.c                                           | 145 |
| RAM                                               |     |
| execution                                         | 83  |
| initializers copied from ROM                      | 52  |
| saving memory                                     | 224 |
| RAMCODE21 (segment)                               | 379 |
| RAMCODE21_ID (segment)                            | 380 |

|                                                   |         |
|---------------------------------------------------|---------|
| RAMCODE32 (segment)                               | 380     |
| RAMCODE32_ID (segment)                            | 380     |
| __ramfunc (extended keyword)                      | 83, 302 |
| range errors, in linker                           | 117     |
| __read (library function)                         | 141     |
| customizing                                       | 137     |
| read formatter, selecting                         | 125     |
| reading guidelines                                | 29      |
| reading, recommended                              | 33      |
| __read_TLB_entry (intrinsic function)             | 340     |
| realloc (library function)                        | 69      |
| implementation-defined behavior in C89            | 416     |
| <i>See also</i> heap                              |         |
| recursive functions                               |         |
| avoiding                                          | 224     |
| storing data on stack                             | 68      |
| reentrancy (DLIB)                                 | 356     |
| reference information, typographic convention     | 34      |
| register keyword, implementation-defined behavior | 396     |
| register parameters                               | 173–174 |
| registered trademarks                             | 2       |
| registers                                         |         |
| assigning to parameters                           | 174     |
| callee-save, stored on stack                      | 68      |
| for function returns                              | 175     |
| implementation-defined behavior in C89            | 411     |
| in assembler-level routines                       | 170     |
| preserved                                         | 172     |
| scratch                                           | 172     |
| reinterpret_cast (cast operator)                  | 196     |
| --relaxed_fp (compiler option)                    | 269     |
| remark (diagnostic message)                       | 235     |
| classifying for compiler                          | 251     |
| enabling in compiler                              | 269     |
| --remarks (compiler option)                       | 269     |
| remove (library function)                         | 141     |
| implementation-defined behavior                   | 402     |
| implementation-defined behavior in C89 (DLIB)     | 415     |
| remquo, magnitude of                              | 400     |

rename (library function) . . . . . 141  
     implementation-defined behavior. . . . . 402  
     implementation-defined behavior in C89 (DLIB) . . . . 415  
 \_\_ReportAssert (library function) . . . . . 148  
 required (pragma directive) . . . . . 322  
 --require\_prototypes (compiler option) . . . . . 270  
 RESET (segment) . . . . . 381  
 RESETCODE (segment) . . . . . 381  
 restrict keyword, enabling . . . . . 256  
 return address register, considerations . . . . . 172  
 return addresses . . . . . 176  
 return values, from functions . . . . . 175  
 Ritchie, Dennis M. . . . . 33  
 rmw, enabling block of instructions . . . . . 56, 244  
 \_\_root (extended keyword) . . . . . 303  
 round (ETSI macro) . . . . . 333  
 routines, time-critical . . . . . 159, 188, 329  
 rtmodel (assembler directive) . . . . . 155  
 rtmodel (pragma directive) . . . . . 323  
 rtti support, missing from STL . . . . . 197  
 \_\_rt\_version (runtime model attribute) . . . . . 156  
 runtime environment  
     DLIB . . . . . 119  
     setting options for . . . . . 58  
     setting up (DLIB) . . . . . 120  
 runtime libraries (DLIB)  
     introduction . . . . . 355  
     customizing system startup code . . . . . 135  
     customizing without rebuilding . . . . . 122  
     filename syntax . . . . . 122  
     overriding modules in . . . . . 129  
     using prebuilt . . . . . 121  
 runtime library  
     setting up from command line . . . . . 57  
     setting up from IDE . . . . . 57  
 runtime model attributes . . . . . 154  
 runtime model definitions . . . . . 323  
 runtime type information, missing from Embedded C++ . 196

## S

\_\_scall (extended keyword) . . . . . 303  
 scall functions . . . . . 78  
 scanf (library function)  
     choosing formatter (DLIB) . . . . . 124  
     configuration symbols . . . . . 139  
     implementation-defined behavior. . . . . 403  
     implementation-defined behavior in C89 (DLIB) . . . . 416  
 \_\_scanf\_args (pragma directive) . . . . . 324  
 scheduling (compiler transformation) . . . . . 223  
     disabling . . . . . 262  
 scratch registers . . . . . 172  
 \_\_search\_TLB\_entry (intrinsic function) . . . . . 340  
 section (pragma directive) . . . . . 324  
 segment group name . . . . . 94  
 segment map, in linker map file . . . . . 118  
 segment memory types, in XLINK . . . . . 90  
 segment (pragma directive) . . . . . 324  
 segments . . . . . 365  
     allocation of . . . . . 92  
     CSTACK, example . . . . . 133  
     declaring (#pragma segment) . . . . . 325  
     definition of . . . . . 90  
     located data . . . . . 108  
     naming . . . . . 94  
     packing in memory . . . . . 107  
     placing in sequence . . . . . 107  
     SSTACK, example . . . . . 133  
     summary . . . . . 365  
     too long for address range . . . . . 117  
     too long, in linker . . . . . 117  
 \_\_segment\_begin (extended operator) . . . . . 191  
 \_\_segment\_end (extended operator) . . . . . 191  
 \_\_segment\_size (extended operator) . . . . . 191  
 segment-translated systems . . . . . 114  
 semaphores  
     C example . . . . . 79  
     C++ example . . . . . 81

|                                             |         |
|---------------------------------------------|---------|
| operations on                               | 298     |
| set (STL header file)                       | 360     |
| setjmp.h (library header file)              | 358     |
| setlocale (library function)                | 143     |
| settings, basic for project configuration   | 54      |
| __set_system_register (intrinsic function)  | 341     |
| __set_interrupt_state (intrinsic function)  | 341     |
| __set_status_flag (intrinsic function)      | 341     |
| __set_system_register (intrinsic function)  | 342     |
| __set_suser_context (intrinsic function)    | 342     |
| severity level, of diagnostic messages      | 235     |
| specifying                                  | 236     |
| <b>SFR</b>                                  |         |
| accessing special function registers        | 227     |
| declaring extern special function registers | 216     |
| shadow_registers (pragma directive)         | 325     |
| shared object                               | 234     |
| shl (ETSI macro)                            | 333     |
| short (data type)                           | 276     |
| shr (ETSI macro)                            | 333     |
| shr_r (ETSI macro)                          | 333     |
| signal (library function)                   |         |
| configuring support for                     | 145     |
| implementation-defined behavior             | 400     |
| implementation-defined behavior in C89      | 414     |
| signals, implementation-defined behavior    | 392     |
| at system startup                           | 392     |
| signal.c                                    | 145     |
| signal.h (library header file)              | 358     |
| signed char (data type)                     | 276–277 |
| specifying                                  | 246     |
| signed int (data type)                      | 276     |
| signed long long (data type)                | 276     |
| signed long (data type)                     | 276     |
| signed short (data type)                    | 276     |
| signed values, avoiding                     | 211     |
| __signed_saturate (intrinsic function)      | 343     |
| --silent (compiler option)                  | 270     |
| silent operation, specifying in compiler    | 270     |

|                                                          |          |
|----------------------------------------------------------|----------|
| simd, enabling block of instructions                     | 56, 245  |
| sin (library function)                                   | 356      |
| sin (library routine)                                    | 146, 148 |
| sinf (library routine)                                   | 147–148  |
| sinl (library routine)                                   | 147–148  |
| 64-bits (floating-point format)                          | 280      |
| size (in stack usage control file)                       | 389      |
| size_t (integer type)                                    | 282      |
| skeleton code, creating for assembler language interface | 167      |
| SLEEP (assembler instruction)                            | 343      |
| __sleep (intrinsic function)                             | 343      |
| slist (STL header file)                                  | 360      |
| source files, list all referred                          | 257      |
| space characters, implementation-defined behavior        | 401      |
| special function registers (SFR)                         | 227      |
| special function types                                   | 73       |
| sprintf (library function)                               | 123      |
| choosing formatter                                       | 123      |
| sscanf (library function)                                |          |
| choosing formatter (DLIB)                                | 124      |
| <b>SSTACK (segment)</b>                                  |          |
| example                                                  | 133      |
| <i>See also</i> stack                                    |          |
| sstream (library header file)                            | 359      |
| stack                                                    | 68       |
| advantages and problems using                            | 68       |
| cleaning after function return                           | 175      |
| contents of                                              | 68       |
| internal data in supervisor mode                         | 381      |
| layout                                                   | 174      |
| saving space                                             | 224      |
| setting up                                               | 109      |
| size                                                     | 205      |
| stack initialization                                     | 133      |
| stack parameters                                         | 173–174  |
| stack pointer                                            | 68       |
| stack pointer register, considerations                   | 172      |
| stack segments                                           |          |
| CSTACK                                                   | 369      |

- placing . . . . . 109
- stack (STL header file) . . . . . 360
- stack-size (in stack usage control file). . . . . 389
- Standard C . . . . . 256
  - library compliance with . . . . . 355
  - specifying strict usage . . . . . 270
- standard error
  - redirecting in compiler . . . . . 266
  - See also diagnostic messages . . . . . 234
- standard input . . . . . 137
- standard output . . . . . 137
  - specifying in compiler . . . . . 266
- standard template library (STL)
  - in C++ . . . . . 359
  - in Extended EC++ . . . . . 196, 200
  - missing from Embedded C++ . . . . . 196
- startup code. . . . . 111
  - cstartup . . . . . 135
  - placement of . . . . . 111
- startup system. *See* system startup
- statements, implementation-defined behavior in C89 . . . . 412
- static analysis
  - documentation for . . . . . 32
- static clustering (compiler transformation) . . . . . 223
- static data, in configuration file. . . . . 108
- static overlay . . . . . 178
- static variables . . . . . 60
  - taking the address of . . . . . 224
- static\_assert() . . . . . 190
- static\_cast (cast operator) . . . . . 196
- status flags for floating-point . . . . . 361
- std namespace, missing from EC++
  - and Extended EC++ . . . . . 200
- stdarg.h (library header file) . . . . . 358
- stdbool.h (library header file) . . . . . 276, 358
- \_\_STDC\_\_ (predefined symbol) . . . . . 352
- STDC CX\_LIMITED\_RANGE (pragma directive) . . . . 326
- STDC FENV\_ACCESS (pragma directive) . . . . . 326
- STDC FP\_CONTRACT (pragma directive) . . . . . 327
- \_\_STDC\_VERSION\_\_ (predefined symbol) . . . . . 352
- stddef.h (library header file) . . . . . 277, 358
- stderr . . . . . 141, 266
- stdin . . . . . 141
  - implementation-defined behavior in C89 (DLIB) . . . . 415
- stdin and stdout, redirecting to C-SPY window . . . . . 148
- stdint.h (library header file). . . . . 358, 360
- stdio.h (library header file) . . . . . 358
- stdio.h, additional C functionality . . . . . 361
- stdlib.h (library header file). . . . . 358
- stdout . . . . . 141, 266
  - implementation-defined behavior. . . . . 401
  - implementation-defined behavior in C89 (DLIB) . . . . 414
- Steele, Guy L. . . . . 33
- STL . . . . . 200
- \_\_store\_conditional (intrinsic function) . . . . . 343
- strcasecmp, in string.h . . . . . 362
- strdup, in string.h . . . . . 362
- streambuf (library header file). . . . . 359
- streams
  - implementation-defined behavior. . . . . 392
  - supported in Embedded C++ . . . . . 196
- strerror (library function), implementation-defined behavior . . . . . 406
- strerror (library function), implementation-defined behavior in C89 (DLIB) . . . . . 417
- strict (compiler option). . . . . 270
- string (library header file) . . . . . 359
- strings, supported in Embedded C++ . . . . . 196
- string.h (library header file) . . . . . 358
- string.h, additional C functionality . . . . . 362
- strncasecmp, in string.h. . . . . 362
- strlen, in string.h . . . . . 362
- Stroustrup, Bjarne . . . . . 33
- strstream (library header file) . . . . . 359
- strtod (library function), configuring support for . . . . . 146
- structure types
  - alignment . . . . . 283
  - layout of. . . . . 283
  - packed . . . . . 283

|                                                              |             |
|--------------------------------------------------------------|-------------|
| structures                                                   |             |
| accessing using a pointer                                    | 180         |
| aligning                                                     | 321         |
| anonymous                                                    | 190, 213    |
| implementation-defined behavior                              | 397         |
| implementation-defined behavior in C89                       | 411         |
| packing and unpacking                                        | 212         |
| placing in memory type                                       | 65          |
| sub (ETSI macro)                                             | 333         |
| subnormal numbers                                            | 281         |
| supervisor mode                                              | 303         |
| entering by using <code>__scall</code>                       | 78          |
| support, technical                                           | 236         |
| Sutter, Herb                                                 | 33          |
| <code>__swap_bytes</code> (intrinsic function)               | 344         |
| <code>__swap_bytes_in_halfwords</code> (intrinsic function)  | 344         |
| <code>__swap_halfwords</code> (intrinsic function)           | 344         |
| switch tables                                                | 112         |
| SWITCH (segment)                                             | 382         |
| symbols                                                      |             |
| anonymous, creating                                          | 187         |
| including in output                                          | 323         |
| listing in linker map file                                   | 118         |
| overview of predefined                                       | 43          |
| preprocessor, defining                                       | 248         |
| <code>__synchronize_write_buffer</code> (intrinsic function) | 345         |
| syntax                                                       |             |
| command line options                                         | 237         |
| extended keywords                                            | 63, 290–292 |
| invoking compiler                                            | 231         |
| <code>__sysreg</code> (extended keyword)                     | 303         |
| <code>SYSREG_AC</code> (segment)                             | 382         |
| system function, implementation-defined behavior             | 393, 403    |
| system locks interface                                       | 151         |
| system startup                                               |             |
| customizing                                                  | 135         |
| DLIB                                                         | 132         |
| initialization phase                                         | 50          |

|                                                     |          |
|-----------------------------------------------------|----------|
| system termination                                  |          |
| C-SPY interface to                                  | 135      |
| DLIB                                                | 134      |
| system (library function)                           |          |
| configuring support for                             | 144      |
| implementation-defined behavior in C89 (DLIB)       | 416      |
| <code>system_include</code> (pragma directive)      | 399, 413 |
| <code>--system_include_dir</code> (compiler option) | 271      |

## T

|                                                               |          |
|---------------------------------------------------------------|----------|
| <code>tan</code> (library function)                           | 356      |
| <code>tan</code> (library routine)                            | 146, 148 |
| <code>tanf</code> (library routine)                           | 147–148  |
| <code>tanl</code> (library routine)                           | 147–148  |
| technical support, IAR Systems                                | 236      |
| template support                                              |          |
| in Extended EC++                                              | 196, 199 |
| missing from Embedded C++                                     | 196      |
| Terminal I/O window                                           |          |
| making available (DLIB)                                       | 127      |
| not supported when                                            | 130      |
| terminal I/O, debugger runtime interface for                  | 126      |
| terminal output, speeding up                                  | 127      |
| termination of system. <i>See</i> system termination          |          |
| termination status, implementation-defined behavior           | 403      |
| terminology                                                   | 34       |
| <code>__test_status_flag</code> (intrinsic function)          | 345      |
| <code>tgmath.h</code> (library header file)                   | 358      |
| 32-bits (floating-point format)                               | 280      |
| <code>this</code> (pointer)                                   | 169      |
| threaded environment                                          | 149      |
| thread-local storage                                          | 151      |
| <code>__TIME__</code> (predefined symbol)                     | 352      |
| time zone (library function)                                  |          |
| implementation-defined behavior in C89                        | 417      |
| time zone (library function), implementation-defined behavior | 403      |
| <code>__TIMESTAMP__</code> (predefined symbol)                | 352      |



time-critical routines . . . . . 159, 188, 329

time.c . . . . . 145

time.h (library header file) . . . . . 358

    additional C functionality . . . . . 362

time32 (library function), configuring support for . . . . . 145

time64 (library function), configuring support for . . . . . 145

\_\_tiny (extended keyword) . . . . . 281

tips, programming . . . . . 223

TLS handling . . . . . 151

tools icon, in this guide . . . . . 34

TRACEBUFFER (segment) . . . . . 382

trademarks . . . . . 2

transformations, compiler . . . . . 218

translation

    implementation-defined behavior . . . . . 391

    implementation-defined behavior in C89 . . . . . 407

trap vectors, specifying with pragma directive . . . . . 328

type attributes . . . . . 289

    specifying . . . . . 327

type definitions, used for specifying memory storage . . . . . 65

type information, omitting . . . . . 266

type qualifiers

    const and volatile . . . . . 284

    implementation-defined behavior . . . . . 397

    implementation-defined behavior in C89 . . . . . 411

typedefs

    excluding from diagnostics . . . . . 264

    repeated . . . . . 192

type\_attribute (pragma directive) . . . . . 327

type-based alias analysis (compiler transformation) . . . . . 222

    disabling . . . . . 264

type-safe memory management . . . . . 195

typographic conventions . . . . . 34

## U

### UBROF

    format of linkable object files . . . . . 233

    specifying, example of . . . . . 54

uchar.h (library header file) . . . . . 358

uintptr\_t (integer type) . . . . . 282

\_\_unaligned\_word\_access (runtime model attribute) . . . . . 156

--unaligned\_word\_access (compiler option) . . . . . 271

underflow errors, implementation-defined behavior . . . . . 400

underflow range errors,

    implementation-defined behavior in C89 . . . . . 414

\_\_ungetchar, in stdio.h . . . . . 361

unions

    anonymous . . . . . 190, 213

    implementation-defined behavior . . . . . 397

    implementation-defined behavior in C89 . . . . . 411

universal character names, implementation-defined

    behavior . . . . . 398

unsigned char (data type) . . . . . 276–277

    changing to signed char . . . . . 246

unsigned int (data type) . . . . . 276

unsigned long long (data type) . . . . . 276

unsigned long (data type) . . . . . 276

unsigned short (data type) . . . . . 276

\_\_unsigned\_saturate (intrinsic function) . . . . . 345

--use\_c++\_inline (compiler option) . . . . . 272

utility (STL header file) . . . . . 360

## V

variable type information, omitting in object output . . . . . 266

variables

    auto . . . . . 68

    defined inside a function . . . . . 68

    global

        accessing . . . . . 180

        placement in memory . . . . . 60

    hints for choosing . . . . . 224

    local. *See* auto variables

    non-initialized . . . . . 228

    omitting type info . . . . . 266

    placing at absolute addresses . . . . . 217

    placing in named segments . . . . . 217

|                                                  |     |
|--------------------------------------------------|-----|
| static                                           |     |
| placement in memory                              | 60  |
| taking the address of                            | 224 |
| --variable_enum_size (compiler option)           | 272 |
| variadic macros                                  | 191 |
| vector (pragma directive)                        | 328 |
| vector (STL header file)                         | 360 |
| version                                          |     |
| compiler subversion number                       | 352 |
| identifying C standard in use (__STDC_VERSION__) | 352 |
| of compiler (__VER__)                            | 353 |
| of this guide                                    | 2   |
| --vla (compiler option)                          | 272 |
| void, pointers to                                | 192 |
| volatile                                         |     |
| and const, declaring objects                     | 286 |
| declaring objects                                | 284 |
| protecting simultaneously accesses variables     | 226 |
| rules for access                                 | 285 |

## W

|                                                         |          |
|---------------------------------------------------------|----------|
| #warning message (preprocessor extension)               | 353      |
| warnings                                                | 235      |
| classifying in compiler                                 | 252      |
| disabling in compiler                                   | 265      |
| exit code in compiler                                   | 273      |
| warnings icon, in this guide                            | 35       |
| warnings (pragma directive)                             | 399, 413 |
| --warnings_affect_exit_code (compiler option)           | 234, 273 |
| --warnings_are_errors (compiler option)                 | 273      |
| --warn_about_c_style_casts (compiler option)            | 273      |
| wchar_t (data type), adding support for in C            | 277      |
| wchar.h (library header file)                           | 358, 361 |
| wctype.h (library header file)                          | 358      |
| web sites, recommended                                  | 33       |
| white-space characters, implementation-defined behavior | 391      |
| With I/O emulation modules (linker option), using       | 148      |

|                                          |     |
|------------------------------------------|-----|
| __write (library function)               | 141 |
| customizing                              | 137 |
| __write_array, in stdio.h                | 361 |
| __write_buffered (DLIB library function) | 127 |
| __write_TLB_entry (intrinsic function)   | 346 |

## X

|                            |     |
|----------------------------|-----|
| XLINK errors               |     |
| range error                | 117 |
| segment too long           | 117 |
| XLINK segment memory types | 90  |
| XLINK. <i>See</i> linker   |     |
| xreportassert.c            | 148 |

# Symbols

|                                                              |     |
|--------------------------------------------------------------|-----|
| __Exit (library function)                                    | 134 |
| __exit (library function)                                    | 134 |
| __acall (extended keyword)                                   | 294 |
| __ALIGNOF__ (operator)                                       | 190 |
| __asm (language extension)                                   | 162 |
| __assignment_by_bitwise_copy_allowed, symbol used in library | 362 |
| __BASE_FILE__ (predefined symbol)                            | 348 |
| __bit_reverse (intrinsic function)                           | 333 |
| __BREAKPOINT (intrinsic function)                            | 333 |
| __BUILD_NUMBER__ (predefined symbol)                         | 348 |
| __cache_control (intrinsic function)                         | 334 |
| __clear_status_flag (intrinsic function)                     | 334 |
| __close (library function)                                   | 141 |
| __code_model (runtime model attribute)                       | 156 |
| __CODE_MODEL__ (predefined symbol)                           | 348 |
| __code, symbol used in library                               | 362 |
| __code21 (extended keyword)                                  | 294 |
| __code32 (extended keyword)                                  | 294 |
| __constrange(), symbol used in library                       | 363 |

- \_\_construction\_by\_bitwise\_copy\_allowed, symbol used in library . . . . . 363
- \_\_COP (intrinsic function) . . . . . 334
- \_\_COP\_get\_registers (intrinsic function) . . . . . 335
- \_\_COP\_get\_register32 (intrinsic function) . . . . . 335
- \_\_COP\_get\_register64 (intrinsic function) . . . . . 335
- \_\_COP\_set\_registers (intrinsic function) . . . . . 336
- \_\_COP\_set\_register32 (intrinsic function) . . . . . 336
- \_\_COP\_set\_register64 (intrinsic function) . . . . . 336
- \_\_core (runtime model attribute) . . . . . 156
- \_\_CORE\_REVISION\_\_ (predefined symbol) . . . . . 348
- \_\_CORE\_\_ (predefined symbol) . . . . . 348
- \_\_COUNTER\_\_ (predefined symbol) . . . . . 348
- \_\_count\_leading\_zeros (intrinsic function) . . . . . 336
- \_\_count\_trailing\_zeros (intrinsic function) . . . . . 337
- \_\_cplusplus (predefined symbol) . . . . . 349
- \_\_data\_model (runtime model attribute) . . . . . 156
- \_\_DATA\_MODEL\_\_ (predefined symbol) . . . . . 349
- \_\_data, symbol used in library . . . . . 362
- \_\_data17 (extended keyword) . . . . . 295
- \_\_data21 (extended keyword) . . . . . 295
- \_\_data32 (extended keyword) . . . . . 296
- \_\_DATE\_\_ (predefined symbol) . . . . . 349
- \_\_dbgreg (extended keyword) . . . . . 296
- \_\_DEFAULT\_CODE\_SEGMENT\_\_ (predefined symbol) 349
- \_\_DEFAULT\_CONST\_SEGMENT\_\_ (predefined symbol) . . . . . 349
- \_\_DEFAULT\_DATA\_SEGMENT\_\_ (predefined symbol) 349
- \_\_disable\_interrupt (intrinsic function) . . . . . 337
- \_\_DLIB\_FILE\_DESCRIPTOR (configuration symbol) . . 141
- \_\_embedded\_cplusplus (predefined symbol) . . . . . 349
- \_\_enable\_interrupt (intrinsic function) . . . . . 337
- \_\_enum\_size (runtime model attribute) . . . . . 156
- \_\_exception (extended keyword) . . . . . 296
- \_\_exchange\_memory (intrinsic function) . . . . . 337
- \_\_exit (library function) . . . . . 134
- \_\_far (extended keyword) . . . . . 281
- \_\_farfunc (function pointer) . . . . . 281
- \_\_FILE\_\_ (predefined symbol) . . . . . 350
- \_\_flashvault (extended keyword) . . . . . 297
- \_\_flashvault\_impl (extended keyword) . . . . . 297
- \_\_FUNCTION\_\_ (predefined symbol) . . . . . 194, 350
- \_\_func\_\_ (predefined symbol) . . . . . 194, 350
- \_\_gets, in stdio.h. . . . . 361
- \_\_get\_debug\_register (intrinsic function) . . . . . 338
- \_\_get\_interrupt\_state (intrinsic function) . . . . . 338
- \_\_get\_system\_register (intrinsic function) . . . . . 339
- \_\_get\_user\_context (intrinsic function) . . . . . 339
- \_\_has\_constructor, symbol used in library . . . . . 363
- \_\_has\_destructor, symbol used in library . . . . . 363
- \_\_HAS\_DSP\_INSTRUCTIONS\_\_ (predefined symbol) . 350
- \_\_HAS\_FPU\_INSTRUCTIONS\_\_ (predefined symbol) . 350
- \_\_HAS\_RMW\_INSTRUCTIONS\_\_ (predefined symbol) 351
- \_\_HAS\_SIMD\_INSTRUCTIONS\_\_ (predefined symbol) 351
- \_\_iar\_cos\_accurate (library routine) . . . . . 148
- \_\_iar\_cos\_accuratef (library routine) . . . . . 148
- \_\_iar\_cos\_accuratel (library routine) . . . . . 148
- \_\_iar\_cos\_small (library routine) . . . . . 146
- \_\_iar\_cos\_smallf (library routine) . . . . . 147
- \_\_iar\_cos\_smallll (library routine) . . . . . 147
- \_\_IAR\_DLIB\_PERTHREAD\_INIT\_SIZE (macro) . . . . 152
- \_\_IAR\_DLIB\_PERTHREAD\_SIZE (macro) . . . . . 152
- \_\_IAR\_DLIB\_PERTHREAD\_SYMBOL\_OFFSET (symbolptr) . . . . . 152
- \_\_iar\_exp\_small (library routine) . . . . . 146
- \_\_iar\_exp\_smallf (library routine) . . . . . 147
- \_\_iar\_exp\_smallll (library routine) . . . . . 147
- \_\_iar\_log\_small (library routine) . . . . . 146
- \_\_iar\_log\_smallf (library routine) . . . . . 147
- \_\_iar\_log\_smallll (library routine) . . . . . 147
- \_\_iar\_log10\_small (library routine) . . . . . 146
- \_\_iar\_log10\_smallf (library routine) . . . . . 147
- \_\_iar\_log10\_smallll (library routine) . . . . . 147
- \_\_iar\_Pow (library routine) . . . . . 148
- \_\_iar\_Powf (library routine) . . . . . 148
- \_\_iar\_Powl (library routine) . . . . . 148
- \_\_iar\_Pow\_accurate (library routine) . . . . . 148
- \_\_iar\_Pow\_accuratef (library routine) . . . . . 148
- \_\_iar\_Pow\_accuratel (library routine) . . . . . 148

|                                                                        |          |                                                                   |          |
|------------------------------------------------------------------------|----------|-------------------------------------------------------------------|----------|
| <code>__iar_Pow_accuratel</code> (library routine) . . . . .           | 148      | <code>__memory_of</code>                                          |          |
| <code>__iar_pow_accuratel</code> (library routine) . . . . .           | 148      | symbol used in library . . . . .                                  | 363      |
| <code>__iar_pow_small</code> (library routine) . . . . .               | 146      | <code>__min</code> (intrinsic function) . . . . .                 | 339      |
| <code>__iar_pow_smallf</code> (library routine) . . . . .              | 147      | <code>__monitor</code> (extended keyword) . . . . .               | 298      |
| <code>__iar_pow_smallll</code> (library routine) . . . . .             | 147      | <code>__near</code> (extended keyword) . . . . .                  | 281      |
| <code>__iar_program_start</code> (label) . . . . .                     | 132      | <code>__nearfunc</code> (function pointer) . . . . .              | 281      |
| <code>__iar_Sin</code> (library routine) . . . . .                     | 146, 148 | <code>__nested</code> (extended keyword) . . . . .                | 299      |
| <code>__iar_Sinf</code> (library routine) . . . . .                    | 147–148  | <code>__noreturn</code> (extended keyword) . . . . .              | 301      |
| <code>__iar_Sinl</code> (library routine) . . . . .                    | 147–148  | <code>__no_alloc</code> (extended keyword) . . . . .              | 299      |
| <code>__iar_Sin_accurate</code> (library routine) . . . . .            | 148      | <code>__no_alloc_str</code> (operator) . . . . .                  | 300      |
| <code>__iar_sin_accurate</code> (library routine) . . . . .            | 148      | <code>__no_alloc_str16</code> (operator) . . . . .                | 300      |
| <code>__iar_Sin_accuratef</code> (library routine) . . . . .           | 148      | <code>__no_alloc16</code> (extended keyword) . . . . .            | 299      |
| <code>__iar_sin_accuratef</code> (library routine) . . . . .           | 148      | <code>__no_init</code> (extended keyword) . . . . .               | 228, 301 |
| <code>__iar_Sin_accuratel</code> (library routine) . . . . .           | 148      | <code>__no_operation</code> (intrinsic function) . . . . .        | 340      |
| <code>__iar_sin_accuratel</code> (library routine) . . . . .           | 148      | <code>__open</code> (library function) . . . . .                  | 141      |
| <code>__iar_Sin_small</code> (library routine) . . . . .               | 146      | <code>__packed</code> (extended keyword) . . . . .                | 301      |
| <code>__iar_sin_small</code> (library routine) . . . . .               | 146      | <code>__PART__</code> (predefined symbol) . . . . .               | 351      |
| <code>__iar_Sin_smallf</code> (library routine) . . . . .              | 147      | <code>__prefetch_cache</code> (intrinsic function) . . . . .      | 340      |
| <code>__iar_sin_smallf</code> (library routine) . . . . .              | 147      | <code>__PRETTY_FUNCTION__</code> (predefined symbol) . . . . .    | 352      |
| <code>__iar_Sin_smallll</code> (library routine) . . . . .             | 147      | <code>__printf_args</code> (pragma directive) . . . . .           | 322      |
| <code>__iar_sin_smallll</code> (library routine) . . . . .             | 147      | <code>__program_start</code> (label) . . . . .                    | 132      |
| <code>__IAR_SYSTEMS_ICC__</code> (predefined symbol) . . . . .         | 351      | <code>__ramfunc</code> (extended keyword) . . . . .               | 302      |
| <code>__iar_tan_accurate</code> (library routine) . . . . .            | 148      | executing in RAM . . . . .                                        | 83       |
| <code>__iar_tan_accuratef</code> (library routine) . . . . .           | 148      | <code>__read</code> (library function) . . . . .                  | 141      |
| <code>__iar_tan_accuratel</code> (library routine) . . . . .           | 148      | customizing . . . . .                                             | 137      |
| <code>__iar_tan_small</code> (library routine) . . . . .               | 146      | <code>__read_TLB_entry</code> (intrinsic function) . . . . .      | 340      |
| <code>__iar_tan_smallf</code> (library routine) . . . . .              | 147      | <code>__ReportAssert</code> (library function) . . . . .          | 148      |
| <code>__iar_tan_smallll</code> (library routine) . . . . .             | 147      | <code>__root</code> (extended keyword) . . . . .                  | 303      |
| <code>__ICCAVR32__</code> (predefined symbol) . . . . .                | 351      | <code>__rt_version</code> (runtime model attribute) . . . . .     | 156      |
| <code>__imported</code> (extended keyword) . . . . .                   | 298      | <code>__scall</code> (extended keyword) . . . . .                 | 303      |
| <code>__init_ihandler</code> , called by system startup code . . . . . | 132      | <code>__scanf_args</code> (pragma directive) . . . . .            | 324      |
| <code>__interrupt</code> (extended keyword) . . . . .                  | 74, 298  | <code>__search_TLB_entry</code> (intrinsic function) . . . . .    | 340      |
| <code>__intrinsic</code> (extended keyword) . . . . .                  | 298      | <code>__segment_begin</code> (extended operator) . . . . .        | 191      |
| <code>__LINE__</code> (predefined symbol) . . . . .                    | 351      | <code>__segment_end</code> (extended operator) . . . . .          | 191      |
| <code>__low_level_init</code> . . . . .                                | 132      | <code>__segment_size</code> (extended operator) . . . . .         | 191      |
| initialization phase . . . . .                                         | 50       | <code>__set_debug_register</code> (intrinsic function) . . . . .  | 341      |
| <code>__low_level_init</code> , customizing . . . . .                  | 135      | <code>__set_interrupt_state</code> (intrinsic function) . . . . . | 341      |
| <code>__lseek</code> (library function) . . . . .                      | 141      | <code>__set_status_flag</code> (intrinsic function) . . . . .     | 341      |
| <code>__max</code> (intrinsic function) . . . . .                      | 339      | <code>__set_system_register</code> (intrinsic function) . . . . . | 342      |

- \_\_set\_user\_context (intrinsic function) . . . . . 342
- \_\_signed\_saturate (intrinsic function) . . . . . 343
- \_\_sleep (intrinsic function) . . . . . 343
- \_\_STDC\_VERSION\_\_ (predefined symbol) . . . . . 352
- \_\_STDC\_\_ (predefined symbol) . . . . . 352
- \_\_store\_conditional (intrinsic function) . . . . . 343
- \_\_swap\_bytes (intrinsic function) . . . . . 344
- \_\_swap\_bytes\_in\_halfwords (intrinsic function) . . . . . 344
- \_\_swap\_halfwords (intrinsic function) . . . . . 344
- \_\_synchronize\_write\_buffer (intrinsic function) . . . . . 345
- \_\_sysreg (extended keyword) . . . . . 303
- \_\_test\_status\_flag (intrinsic function) . . . . . 345
- \_\_TIMESTAMP\_\_ (predefined symbol) . . . . . 352
- \_\_TIME\_\_ (predefined symbol) . . . . . 352
- \_\_tiny (extended keyword) . . . . . 281
- \_\_unaligned\_word
  - access (runtime model attribute) . . . . . 156
- \_\_ungetchar, in stdio.h . . . . . 361
- \_\_unsigned\_saturate (intrinsic function) . . . . . 345
- \_\_VA\_ARGS\_\_ (preprocessor extension) . . . . . 187
- \_\_write (library function) . . . . . 141
  - customizing . . . . . 137
- \_\_write\_array, in stdio.h . . . . . 361
- \_\_write\_buffered (DLIB library function) . . . . . 127
- \_\_write\_TLB\_entry (intrinsic function) . . . . . 346
- D (compiler option) . . . . . 248
- e (compiler option) . . . . . 254
- f (compiler option) . . . . . 256
- I (compiler option) . . . . . 258
- l (compiler option) . . . . . 258
  - for creating skeleton code . . . . . 168
- O (compiler option) . . . . . 265
- o (compiler option) . . . . . 266
- r (compiler option) . . . . . 249
- avr32\_dsp\_instructions (compiler option) . . . . . 243
- avr32\_flashvault (compiler option) . . . . . 243
- avr32\_fpu\_instructions (compiler option) . . . . . 244
- avr32\_rmw\_instructions (compiler option) . . . . . 244
- avr32\_simd\_instructions (compiler option) . . . . . 245
- char\_is\_signed (compiler option) . . . . . 246
- char\_is\_unsigned (compiler option) . . . . . 246
- code\_model (compiler option) . . . . . 246
- cpu (compiler option) . . . . . 247
- cpu\_info (compiler option) . . . . . 248
- c89 (compiler option) . . . . . 245
- data\_model (compiler option) . . . . . 249
- debug (compiler option) . . . . . 249
- dependencies (compiler option) . . . . . 250
- diagnostics\_tables (compiler option) . . . . . 252
- diag\_error (compiler option) . . . . . 251
- diag\_remark (compiler option) . . . . . 251
- diag\_suppress (compiler option) . . . . . 251
- diag\_warning (compiler option) . . . . . 252
- disable\_inline\_asm\_label\_replacement
  - (compiler option) . . . . . 253
- discard\_unused\_publics (compiler option) . . . . . 253
- dlib\_config (compiler option) . . . . . 253
- ec++ (compiler option) . . . . . 255
- eec++ (compiler option) . . . . . 255
- enable\_multibytes (compiler option) . . . . . 255
- enable\_restrict (compiler option) . . . . . 256
- error\_limit (compiler option) . . . . . 256
- fp\_implementation (compiler option) . . . . . 257
- guard\_calls (compiler option) . . . . . 257
- g\_handle\_all\_exceptions, handle unhandled exceptions 76
- g\_init\_all\_ihandlers, handle unhandled interrupts . . . . . 75
- header\_context (compiler option) . . . . . 257
- library\_module (compiler option) . . . . . 259
- macro\_positions\_in\_diagnostics (compiler option) . . . . . 259
- mfc (compiler option) . . . . . 260
- minimize\_constant\_tables (compiler option) . . . . . 260
- misrac\_verbose (compiler option) . . . . . 241
- misrac1998 (compiler option) . . . . . 241
- misrac2004 (compiler option) . . . . . 241
- module\_name (compiler option) . . . . . 260
- no\_clustering (compiler option) . . . . . 261
- no\_code\_motion (compiler option) . . . . . 261
- no\_cse (compiler option) . . . . . 261
- no\_inline (compiler option) . . . . . 262

|                                                          |          |
|----------------------------------------------------------|----------|
| --no_path_in_file_macros (compiler option) . . . . .     | 262      |
| --no_scheduling (compiler option) . . . . .              | 262      |
| --no_size_constraints (compiler option) . . . . .        | 263      |
| --no_static_destruction (compiler option) . . . . .      | 263      |
| --no_system_include (compiler option) . . . . .          | 263      |
| --no_typedefs_in_diagnostics (compiler option) . . . . . | 264      |
| --no_unroll (compiler option) . . . . .                  | 264      |
| --no_warnings (compiler option) . . . . .                | 265      |
| --no_wrap_diagnostics (compiler option) . . . . .        | 265      |
| --omit_types (compiler option) . . . . .                 | 266      |
| --only_stdout (compiler option) . . . . .                | 266      |
| --output (compiler option) . . . . .                     | 266      |
| --pending_instantiations (compiler option) . . . . .     | 267      |
| --predef_macro (compiler option) . . . . .               | 267      |
| --preinclude (compiler option) . . . . .                 | 268      |
| --preprocess (compiler option) . . . . .                 | 268      |
| --relaxed_fp (compiler option) . . . . .                 | 269      |
| --remarks (compiler option) . . . . .                    | 269      |
| --require_prototypes (compiler option) . . . . .         | 270      |
| --silent (compiler option) . . . . .                     | 270      |
| --strict (compiler option) . . . . .                     | 270      |
| --system_include_dir (compiler option) . . . . .         | 271      |
| --unaligned_word_access (compiler option) . . . . .      | 271      |
| --use_cplusplus_inline (compiler option) . . . . .       | 272      |
| --variable_enum_size (compiler option) . . . . .         | 272      |
| --vla (compiler option) . . . . .                        | 272      |
| --warnings_affect_exit_code (compiler option) . . . . .  | 234, 273 |
| --warnings_are_errors (compiler option) . . . . .        | 273      |
| --warn_about_c_style_casts (compiler option) . . . . .   | 273      |
| @ (operator)                                             |          |
| placing at absolute address . . . . .                    | 215      |
| placing in segments . . . . .                            | 217      |
| #include files, specifying . . . . .                     | 232, 258 |
| #warning message (preprocessor extension) . . . . .      | 353      |
| %Z replacement string,                                   |          |
| implementation-defined behavior . . . . .                | 404      |

## Numerics

|                                             |     |
|---------------------------------------------|-----|
| 16-bit pointers, accessing memory . . . . . | 67  |
| 24-bit pointers, accessing memory . . . . . | 67  |
| 32-bits (floating-point format) . . . . .   | 280 |
| 64-bit data types, avoiding . . . . .       | 211 |
| 64-bits (floating-point format) . . . . .   | 280 |