IAR Embedded Workbench®

IAR C/C++ Compiler

Reference Guide

for the Renesas

V850 Microcontroller Family

CV850-9

COPYRIGHT NOTICE

© 1998–2013 IAR Systems AB.

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

DISCLAIMER

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

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

TRADEMARKS

IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, The Code to Success, IAR KickStart Kit, I-jet, I-scope, IAR and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Renesas is a registered trademark of Renesas Electronics Corporation. V850 is a trademark of Renesas Electronics Corporation.

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

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

EDITION NOTICE

Ninth edition: May 2013

Part number: CV850-9

This guide applies to version 4.x of IAR Embedded Workbench® for the Renesas V850 microcontroller family.

Internal reference: M13, Too6.5, csrct2010.1, V_110411, IJOA.

Brief contents

Tables 23

Preface 25

Part 1. Using the compiler 33

Getting started 35

Data storage 43

Functions 53

Placing code and data 67

The DLIB runtime environment 81

Assembler language interface 117

Using C 145

Using C++ 153

Efficient coding for embedded applications 163

Part 2. Reference information 183

External interface details 185

Compiler options 191

Data representation 227

Extended keywords 239

Pragma directives 251

Intrinsic functions 271

The preprocessor 281

Library functions 289

Segment reference 299
Implementation-defined behavior for Standard C 317
Implementation-defined behavior for C89 333
Index 345

Contents

Tables 23

Preface 25

Who should read this guide 25

How to use this guide 25

What this guide contains 25

Part 1. Using the compiler 26

Part 2. Reference information 26

Other documentation 27

User and reference guides 27

The online help system 28

Further reading 28

Web sites 29

Document conventions 29

Typographic conventions 30

Naming conventions 30

Part 1. Using the compiler 33

Getting started 35

IAR language overview 35

Supported V850 devices 36

Building applications—an overview 36

Compiling 36

Linking 37

Basic project configuration 37

Core 38

Data model 38

Code model 38

Optimization for speed and size 39

Runtime environment 39

Special support for embedded systems 40
Extended keywords 41
Pragma directives 41
Predefined symbols 41
Special function types 41
Accessing low-level features 41

Data storage 43

Introduction 43
Different ways to store data 43
Data models 44
Specifying a data model 44
Memory types 45
Near 46
Brel (base-relative) 46
Brel23 (base-relative23) 47
Huge 47
Saddr (Short addressing) 47
Using data memory attributes 47
Structures and memory types 49
More examples 49
C++ and memory types 50
Auto variables—on the stack 50
The stack 51
Dynamic memory on the heap 52
Potential problems 52

Functions 53

Function-related extensions 53
Code models 53
The normal code model 54
The large code model 54
The position-independent code model 54

Primitives for interrupts, concurrency, and OS-related programming ... 55
Interrupt functions ... 55
Trap functions ... 57
Callt functions ... 58
Syscall functions ... 58
Monitor functions ... 59
C++ and special function types ... 62
Inlining functions ... 63
C versus C++ semantics ... 63
Features controlling function inlining ... 64
Position-independent code ... 64
The distance moved ... 65
Calling functions outside the application ... 65
Placing code and data ... 67
Segments and memory ... 67
What is a segment? ... 67
Placing segments in memory ... 68
Customizing the linker configuration file ... 68
Data segments ... 71
Static memory segments ... 71
The stack ... 74
The heap ... 75
Located data ... 76
User-defined segments ... 77
Code segments ... 77
Startup code ... 77
Normal code ... 77
Interrupt vectors and functions ... 77
Trap vectors ... 78
Callt functions ... 78
Syscall functions ... 78
C++ dynamic initialization ... 78

Verifying the linked result of code and data placement 79
Segment too long errors and range errors 79
Linker map file 79

The DLIB runtime environment 81

Introduction to the runtime environment 81
Runtime environment functionality 81
Setting up the runtime environment 82
Using a prebuilt library 83
Choosing a library 83
Customizing a prebuilt library without rebuilding 86
Choosing formatters for printf and scanf 87
Choosing a printf formatter 87
Choosing a scanf formatter 88
Application debug support 89
Including C-SPY debugging support 89
The debug library functionality 90
The C-SPY Terminal I/O window 91
Low-level functions in the debug library 92
Adapting the library for target hardware 92
Library low-level interface 93
Overriding library modules 93
Building and using a customized library 94
Setting up a library project 94
Modifying the library functionality 95
Using a customized library 95
System startup and termination 95
System startup 96
System termination 98
Customizing system initialization 99
__low_level_init 99
Modifying the file cstartup.s85 99
Library configurations 100
Choosing a runtime configuration 100

Standard streams for input and output 101
Implementing low-level character input and output 101
Configuration symbols for printf and scanf 103
Customizing formatting capabilities 104
File input and output 105
Locale 105
Locale support in prebuilt libraries 106
Customizing the locale support 106
Changing locales at runtime 107
Environment interaction 108
The getenv function 108
The system function 108
Signal and raise 109
Time 109
Strtod 110
Math functions 110
Smaller versions 110
More accurate versions 111
Assert 112
Checking module consistency 112
Runtime model attributes 113
Using runtime model attributes 113
Predefined runtime attributes 114

Assembler language interface 117

Mixing C and assembler 117
Intrinsic functions 117
Mixing C and assembler modules 118
Inline assembler 119
Calling assembler routines from C 121
Creating skeleton code 121
Compiling the code 122
Calling assembler routines from C++ 123

Calling convention ... 124
Function declarations ... 124
Using C linkage in C++ source code ... 124
Preserved versus scratch registers ... 125
Function entrance ... 126
Function exit ... 128
Restrictions for special function types ... 129
Examples ... 130
Function directives ... 131
Calling functions ... 131
Assembler instructions used for calling functions ... 131
Memory access methods ... 134
Near memory access methods ... 136
Base-relative access method ... 137
Base-relative23 access method ... 138
Huge access method ... 138
Short addressing access method ... 139
No bit access ... 139
Call frame information ... 139
CFI directives ... 140
Creating assembler source with CFI support ... 140
Using C ... 145
C language overview ... 145
Extensions overview ... 146
Enabling language extensions ... 147
IAR C language extensions ... 147
Extensions for embedded systems programming ... 148
Relaxations to Standard C ... 150
Using C++ ... 153
Overview ... 153
Embedded C++ ... 153
Extended Embedded C++ ... 154
Enabling support for C++ ... 155

EC++ feature descriptions 155
Using IAR attributes with Classes 155
Function types 156
Using static class objects in interrupts 157
Using New handlers 157
Templates 157
Debug support in C-SPY 157
EEC++ feature description 158
Templates 158
Variants of cast operators 158
Mutable 158
Namespace 158
The STD namespace 158
C++ language extensions 159
Efficient coding for embedded applications 163
Selecting data types 163
Using efficient data types 163
Floating-point types 164
Alignment of elements in a structure 164
Anonymous structs and unions 165
Controlling data and function placement in memory 167
Data placement at an absolute location 168
Data and function placement in segments 169
Controlling compiler optimizations 170
Scope for performed optimizations 171
Multi-file compilation units 171
Optimization levels 172
Speed versus size 173
Fine-tuning enabled transformations 173
Register locking and register constants 176
Register locking 176
Register constants 176
Compatibility issues 177

Facilitating good code generation 177
Writing optimization-friendly source code 177
Saving stack space and RAM memory 178
Extending the code span 178
Function prototypes 178
Integer types and bit negation 179
Protecting simultaneously accessed variables 180
Accessing special function registers 181
Non-initialized variables 181

Part 2. Reference information 183

External interface details 185

Invocation syntax 185
Compiler invocation syntax 185
Passing options 185
Environment variables 186
Include file search procedure 186
Compiler output 187
Error return codes 188
Diagnostics 189
Message format 189
Severity levels 189
Setting the severity level 190
Internal error 190

Compiler options 191

Options syntax 191
Types of options 191
Rules for specifying parameters 191
Summary of compiler options 193
Descriptions of compiler options 197
--aggressive_inlining 197
--aggressive_unrolling 197

--allow_misaligned_data_access 198
--c89 198
--char_is_signed 198
--char_is_unsigned 199
--code_model 199
--cpu 199
-D 200
--data_model, -m 200
--debug, -r 201
--dependencies 201
--diag_error 202
--diag_remark 203
--diag_suppress 203
--diag_warning 204
--diagnostics_tables 204
--disable_sld_suppression 204
--discard_unused_publics 205
--dlib_config 205
-e 206
--ec++ 206
--eec++ 206
--enable_multibytes 207
--error_limit 207
-f 208
--fpu 208
--guard_calls 208
--header_context 209
-I 209
-l 209
--library_module 210
--lock_regs 211
--lock_regs_compatibility 211
--macro_positions_in_diagnostics 212
--mfc 212

--migration_preprocessor_extensions 212
--module_name 213
--no_clustering 213
--no_code_motion 214
--no_cross_call 214
--no_cse 214
--no_data_model_rt_attribute 215
--no_inline 215
--no_path_in_file_macros 215
--no_scheduling 216
--no_size_constraints 216
--no_static_destruction 216
--no_system_include 217
--no_tbaa 217
--no_typedefs_in_diagnostics 217
--no_unroll 218
--no_warnings 218
--no_wrap_diagnostics 218
-O 219
--omit_types 219
--only_stdout 220
--output, -o 220
--predef_macros 220
--preinclude 221
--preprocess 221
--public_equ 221
--reg_const 222
--relaxed_fp 222
--remarks 223
--require_prototypes 223
--silent 224
--strict 224
--system_include_dir 224
--use_c++_inline 225

-v 225
--vla 226
--warnings_affect_exit_code 226
--warnings_are_errors 226

Data representation 227

Alignment 227
Alignment on the V850 microcontroller 227
Basic data types 228
Integer types 228
Floating-point types 231
Pointer types 232
Function pointers 232
Data pointers 232
Casting 233
Structure types 233
Alignment 233
General layout 234
Packed structure types 234
Type qualifiers 235
Declaring objects volatile 235
Declaring objects volatile and const 237
Declaring objects const 237
Data types in C++ 237

Extended keywords 239

General syntax rules for extended keywords 239
Type attributes 239
Object attributes 241
Summary of extended keywords 242
Descriptions of extended keywords 243
__brel 243
__brel23 243
__callt 244
__flat 244

__huge 245
__interrupt 245
__intrinsic 246
__monitor 246
__near 246
__no_bit_access 246
__no_init 247
__noreturn 247
__root 247
__saddr 248
__syscall 248
__task 249
__trap 249

Pragma directives 251

Summary of pragma directives 251
Descriptions of pragma directives 253
bitfields 253
constseg 253
data_alignment 254
dataseg 254
diag_default 255
diag_error 255
diag_remark 256
diag_suppress 256
diag_warning 256
error 257
include_alias 257
inline 258
language 258
location 259
message 260
no_epilogue 260
object_attribute 261

optimize ... 261
pack ... 262
__printf_args ... 263
required ... 264
rtmodel ... 264
__scanf_args ... 265
segment ... 265
STDC CX_LIMITED_RANGE ... 266
STDC FENV_ACCESS ... 267
STDC FP_CONTRACT ... 267
type_attribute ... 268
unroll ... 268
vector ... 269

Intrinsic functions ... 271

Summary of intrinsic functions ... 271

Descriptions of intrinsic functions ... 272

__absolute_to_pic ... 272
__code_distance ... 273
__compare_and_exchange_for_interlock ... 273
__disable_interrupt ... 274
__enable_interrupt ... 274
__fpu_sqrt_double ... 274
__fpu_sqrt_float ... 275
__get_interrupt_state ... 275
__get_processor_register ... 276
__halt ... 277
__no_operation ... 277
__pic_to_absolute ... 277
__saturated_add ... 277
__saturated_sub ... 278
__search_ones_left ... 278
__search_ones_right ... 278
__search_zeros_left ... 279

__search_zeros_right 279
__set_interrupt_state 279
__set_processor_register 279
__synchronize_exceptions 280
__synchronize_memory 280
__synchronize_pipeline 280
__upper_mul64 280
The preprocessor 281
Overview of the preprocessor 281
Description of predefined preprocessor symbols 282
__BASE_FILE__ 282
__BUILD_NUMBER__ 282
__CODE_MODEL__ 282
__CORE__ 282
__cplusplus__ 282
__CPU__ 283
__DATA_MODEL__ 283
__DATE__ 283
__embedded_cplusplus 283
__FILE__ 283
__FPU__ 284
__func__ 284
__FUNCTION__ 284
__IAR_SYSTEMS_ICC__ 284
__ICCV850__ 284
__LINE__ 285
__LITTLE_ENDIAN__ 285
__PRETTY_FUNCTION__ 285
__SADDR_ACTIVE__ 285
__STDC__ 285
__STDC_VERSION__ 285
__SUBVERSION__ 286
__TIME__ 286

__VER__ 286
Descriptions of miscellaneous preprocessor extensions 286
NDEBUG 286
#warning message 287
Library functions 289
Library overview 289
Header files 289
Library object files 289
Alternative more accurate library functions 290
Reentrancy 290
The longjmp function 291
IAR DLIB Library 291
C header files 291
C++ header files 292
Library functions as intrinsic functions 295
Added C functionality 295
Symbols used internally by the library 296
Segment reference 299
Summary of segments 299
Descriptions of segments 301
BREL_BASE 301
BREL_CBASE 301
BREL_C 302
BREL_I 302
BREL_ID 302
BREL_N 303
BREL_Z 303
BREL23_C 303
BREL23_I 304
BREL23_ID 304
BREL23_N 304
BREL23_Z 305
CHECKSUM 305

CLTCODE 305
CLTVEC 306
CODE 306
CSTACK 306
CSTART 307
DIFUNCT 307
GLOBAL_AC 307
GLOBAL_AN 307
HEAP 308
HUGE_C 308
HUGE_I 308
HUGE_ID 309
HUGE_N 309
HUGE_Z 309
ICODE 310
INTVEC 310
NEAR_C 310
NEAR_I 310
NEAR_ID 311
NEAR_N 311
NEAR_Z 311
RCODE 312
SADDR_BASE 312
SADDR7_I 312
SADDR7_ID 313
SADDR7_N 313
SADDR7_Z 313
SADDR8_I 314
SADDR8_ID 314
SADDR8_N 314
SADDR8_Z 315
SYSCALLCODE 315
SYSCALLVEC 315
TRAPVEC 316

Implementation-defined behavior for Standard C 317

Descriptions of implementation-defined behavior 317

J.3.1 Translation 317

J.3.2 Environment 318

J.3.3 Identifiers 319

J.3.4 Characters 319

J.3.5 Integers 320

J.3.6 Floating point 321

J.3.7 Arrays and pointers 322

J.3.8 Hints 322

J.3.9 Structures, unions, enumerations, and bitfields 323

J.3.10 Qualifiers 323

J.3.11 Preprocessing directives 323

J.3.12 Library functions 325

J.3.13 Architecture 330

J.4 Locale 330

Implementation-defined behavior for C89 333

Descriptions of implementation-defined behavior 333

Translation 333

Environment 333

Identifiers 334

Characters 334

Integers 335

Floating point 336

Arrays and pointers 336

Registers 337

Structures, unions, enumerations, and bitfields 337

Qualifiers 337

Declarators 338

Statements 338

Preprocessing directives 338

IAR DLIB Library functions 340

Index 345

Tables

1: Typographic conventions used in this guide 30
2: Naming conventions used in this guide 30
3: Data model characteristics 45
4: Memory types and their corresponding memory attributes 48
5: Code models 54
6: Preserved registers in interrupt functions 56
7: XLINK segment memory types 68
8: Memory layout of a target system (example) 69
9: Memory types with corresponding segment groups 72
10: Segment name suffixes 72
11: Prebuilt libraries 84
12: Customizable items 86
13: Formatters for printf 87
14: Formatters for scanf 88
15: Levels of debugging support in runtime libraries 89
16: Functions with special meanings when linked with debug library 92
17: Library configurations 100
18: Descriptions of printf configuration symbols 104
19: Descriptions of scanf configuration symbols 104
20: Low-level I/O files 105
21: Example of runtime model attributes 113
22: Predefined runtime model attributes 114
23: Registers used for passing parameters 127
24: Assembler instructions for accessing memory 134
25: Call frame information resources defined in a names block 140
26: Language extensions 147
27: Compiler optimization levels 172
28: Compiler environment variables 186
29: Error return codes 188
30: Compiler options summary 193
31: Integer types 228

32: Floating-point types 231
33: Extended keywords summary 242
34: Pragma directives summary 251
35: Intrinsic functions summary 271
36: Traditional Standard C header files—DLIB 291
37: C++ header files 293
38: Standard template library header files 293
39: New Standard C header files—DLIB 294
40: Segment summary 299
41: Message returned by strerror()—IAR DLIB library 332
42: Message returned by strerror()—IAR DLIB library 343

Preface

Welcome to the *IAR C/C++ Compiler Reference Guide for V850*. 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 V850 microcontroller and need detailed reference information on how to use the compiler. You should have working knowledge of:

- The architecture and instruction set of the V850 microcontroller. Refer to the documentation from Renesas for information about the V850 microcontroller
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you start using the IAR C/C++ Compiler for V850, you should read *Part 1. Using the compiler* in this guide.

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

If you are new to using the IAR Systems build tools, we recommend that you first study the *IDE Project Management and Building Guide*. This guide contains a product overview, conceptual and user information about the IDE and the IAR C-SPY® Debugger, and corresponding reference information.

What this guide contains

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

PART 1. USING THE COMPILER

- *Getting started* gives the information you need to get started using the compiler for efficiently developing your application.
- *Data storage* describes how to store data in memory, focusing on the different data models and data memory type attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker configuration file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

PART 2. REFERENCE INFORMATION

- *External interface details* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler's diagnostic system works.
- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Extended keywords* gives reference information about each of the V850-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 V850-specific low-level features.

- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior 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. For information about:

- System requirements and information about how to install and register the IAR Systems products, refer to the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, see the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, see the *IDE Project Management and Building Guide.*
- Using the IAR C-SPY® Debugger, see the *C-SPY® Debugging Guide for V850.*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, see the IAR Linker and Library Tools Reference Guide.
- Programming for the IAR Assembler for V850, see the *IAR Assembler Reference Guide for V850.*
- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.

- Porting application code and projects created with a previous version of the IAR Embedded Workbench for V850, see the *IAR Embedded Workbench® Migration Guide*.
- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about debugging using the IAR C-SPY® Debugger
- Information about using the editor
- 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:

- 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 Renesas web site, **www.renesas.com**, that contains information and news about the V850 microcontrollers.
- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.
- The web site of the C standardization working group, **www.open-std.org/jtc1/sc22/wg14**.
- The web site of the C++ Standards Committee, **www.open-std.org/jtc1/sc22/wg21**.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, that contains information about the Embedded C++ standard.

Document conventions

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

When referring to a directory in your product installation, for example `v850\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.n\v850\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
`[option]`	An optional part of a command.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for V850	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for V850	the IDE
IAR C-SPY® Debugger for V850	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for V850	the compiler

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR Assembler™ for V850	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 (Continued)

Part 1. Using the compiler

This part of the *IAR C/C++ Compiler Reference Guide for V850* includes these chapters:

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

Getting started

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

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

Next, the compiler is introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the V850 microcontroller. In the following chapters, these techniques are studied in more detail.

IAR language overview

There are two high-level programming languages you can use with the IAR C/C++ Compiler for V850

- 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 Reference Guide for V850*.

Supported V850 devices

The IAR C/C++ Compiler for V850 supports all devices based on the standard Renesas V850 microcontroller cores: V850, V850E, V850ES, V850E2, V850E2M, and V850E2S.

Building applications—an overview

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

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

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

Below, the process for building an application on the command line is described. For information about how to build an application using the IDE, see the *IDE Project Management and Building Guide*.

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r85` using the default settings:

```
iccv850 myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 37.

LINKING

The IAR XLINK Linker is used for building the final application. Normally, XLINK requires the following information as input:

- One or more object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label
- A linker configuration file that describes the placement of code and data into the memory of the target system
- Information about the output format.

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

```
xlink myfile.r85 myfile2.r85 -s __program_start -f lnk85.xcl
dl85nn0.r85 -o aout.a85 -r
```

In this example, `myfile.r85` and `myfile2.r85` are object files, `lnk85.xcl` is the linker configuration file, and `dl85nn0.r85` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

The IAR XLINK Linker produces output according to your specifications. Choose the output format that suits your purpose. You might want to load the output to a debugger—which means that you need output with debug information. Alternatively, you might want to load the output to a flash loader or a PROM programmer—in which case you need output without debug information, such as Intel hex or Motorola S-records. The option `-F` can be used for specifying the output format. (The default output format is `Intel-extended`.)

Basic 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 V850 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:

- Core
- Data model
- Code model
- Optimization settings

- Runtime environment.

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

CORE

To make the compiler generate optimum code, you should configure it for the V850 microcontroller you are using.

The `--cpu=core` option is used for declaring the specific *CPU core* that is used.

For a list of supported cores, see *Supported V850 devices*, page 36.

In the IDE, choose **Project>Options** and choose an appropriate device from the **Device** drop-down list. The core option will then be automatically set. Note that device-specific configuration files for the linker and the debugger will also be automatically selected.

DATA MODEL

One of the characteristics of the V850 microcontroller is a trade-off in how memory is accessed, between the range from cheap access to small memory areas, up to more expensive access methods that can access any location.

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

- The *Tiny* data model uses near memory for storing data
- The *Small* data model uses brel memory for storing data
- The *Medium* data model uses brel23 memory for storing data
- The *Large* data model uses huge memory for storing data.

All the data models are available with or without support for *short addressing*. For more information about data models, see the chapter *Data storage*. The chapter also covers how to fine-tune the access method for individual variables.

CODE MODEL

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

- The *Normal* code model has an upper limit of 2 Mbytes
- The *Large* code model can access the entire 32-bit address space

- The *Position-Independent* code model has an upper limit of 2 Gbytes, and generates code that can be placed and executed anywhere in memory.

For more information about the code models, see the chapter *Functions*. The chapter also covers how to override the default code model for individual functions.

OPTIMIZATION FOR SPEED AND SIZE

The compiler's optimizer performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, static clustering, 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, which supports Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.

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

The runtime library you choose can be one of the prebuilt libraries, or a library that you customized and built yourself. The IDE provides a library project template that you can use for building your own library version. This gives you full control of the runtime environment. If your project only contains assembler source code, you do not need to choose a runtime library.

For more information about the runtime environment, 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 100, 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 for example look like this:

```
dl85nn1.r85
```

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

In addition to these options you might want to specify any 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 a prebuilt library*, page 83 (DLIB). 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 87 (DLIB).
- The size of the stack and the heap, see *The stack*, page 74, and *The heap*, page 75, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the V850 microcontroller.

EXTENDED KEYWORDS

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

By default, language extensions are enabled in the IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 206 for additional information.

For more information about the extended keywords, see the chapter *Extended keywords*.

PRAGMA DIRECTIVES

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

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

For 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, and the code and data models.

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

SPECIAL FUNCTION TYPES

The special hardware features of the V850 microcontroller family are supported by the compiler's special function types: interrupt, monitor, callt, syscall, task, and trap. You can write a complete application without having to write any of these functions in assembler language.

For more information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 55.

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 117.

Data storage

This chapter gives a brief introduction to the memory layout of the V850 microcontroller and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. For efficient memory usage, the compiler provides a set of data models and data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of data models and memory types are described in relation to pointers, structures, Embedded C++ class objects, and non-initialized memory. Finally, detailed information about data storage on the stack and the heap is provided.

Introduction

The V850 microcontroller has one continuous 4 Gbyte memory space. Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

The compiler can access memory in different ways. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. For more information about this, see *Memory types*, page 45.

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 *Auto variables—on the stack*, page 50.

- 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 44 and *Memory types*, page 45.

- 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 52.

Data models

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

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

The 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 47.

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

SPECIFYING A DATA MODEL

Four data models are implemented: Tiny, Small, Medium, and Large. These data models can also be used with short addressing. Then they are referred to as Tiny with saddr, Small with saddr, Medium with saddr, and Large with saddr. These models are controlled by the `--data_model` option. Each model has a default memory type. If you do not specify a data model option, the compiler will use the Small data model.

Note: The tiny data model cannot be used together with the position-independent *code model*.

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects by explicitly specifying a memory attribute, see *Using data memory attributes*, page 47.

This table summarizes the different data models:

Data model name	Default memory attribute	Placement of data
Tiny	`__near`	Low 32 Kbytes or high 32 Kbytes.
Tiny with saddr	`__near`	As in the tiny data model.
Small (default)	`__brel`	64 Kbytes RAM and 64 Kbytes ROM, anywhere in memory.
Small with saddr	`__brel`	As in the small data model.
Medium	`__brel23`	8 Mbytes in RAM and 8 Mbytes ROM. Only available for the V850E2M core and above.
Medium with saddr	`__brel23`	As in the medium data model. Only available for the V850E2M core and above.
Large	`__huge`	The entire 4 Gbyte of memory.
Large with saddr	`__huge`	As in the huge data model.

Table 3: Data model characteristics

Note: An application module compiled using a data model *with* support for short addressing (saddr) can be linked with modules using the corresponding data model *without* saddr support if no saddr variables are actually used.

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

Use the `--data_model` option to specify the data model for your project; see *--data_model, -m*, page 200.

Memory types

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

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

For example, the memory accessed using near addressing is called near memory.

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

For more information about memory access methods, see *Memory access methods*, page 134.

NEAR

The near memory consists of the low and high 32 Kbytes of memory. In hexadecimal notation, this is the addresses `0x00000000–0x00007FFF` and `0xFFFF8000–0xFFFFFFFF`.

This combination of memory ranges might at first sight seem odd. The explanation, however, is that when an address expression becomes negative, the calculation wraps around. Because the address space on the V850 microcontroller is 32 bits, the address below 0 can be seen as `0xFFFFFFFF`. Hence, an alternative way to see the memory range in the memory accessible is simply ±32 Kbytes around address 0.

Accessing near memory is very efficient, typically only one machine instruction is needed.

BREL (BASE-RELATIVE)

Using base-relative addressing, a 64-Kbyte RAM area and a 64-Kbyte ROM area can be accessed. These brel memory areas can be placed individually at any location in memory.

The name *base-relative* comes from the use of processor registers as base pointers to the memory areas. The RAM area is accessed using the register `R4`, also named `GP` (global pointer) via the label `?BREL_BASE`. The ROM area is accessed using the register `R25` via the label `?BREL_CBASE`.

Access to this type of memory is almost as efficient as accessing near memory.

Because different access methods are used for brel RAM and brel ROM, respectively, a variable declaration must specify whether a RAM or ROM access should be used. In C, this is possible for all variables.

Limitation on const declared objects in C++

In standard C++, a constant variable without constructors can either be placed in ROM if it is initialized with a constant, or in RAM if an expression that must be executed at runtime is used. To solve this ambiguity, the compiler does not allow constant variables without constructors in RAM, only in ROM.

BREL23 (BASE-RELATIVE23)

Using the same base pointers as brel memory, the brel23 memory can access an 8-Mbyte RAM area and an 8-Mbyte ROM area. Brel23 is only available for the V850E2M core and above.

HUGE

The V850 microcontroller has an address space of 4 Gbytes—huge memory. Using this memory type, the data objects can be placed anywhere in memory. Also, unlike the other memory types, there is no limitation on the size of the objects that can be placed in this memory type.

The drawback of the huge memory type is that the code generated to access the memory is larger and also slower than that of any of the other memory types. In addition, the code consumes more processor registers, possibly forcing local variables to be stored on the stack rather than being allocated in registers.

SADDR (SHORT ADDRESSING)

Short addressing can be used for storing variables in a relatively small memory area, 256 bytes, which can be accessed using highly efficient special instructions.

There is a limitation; objects that could be accessed using byte access may only occupy 128 of these bytes. This, of course, includes the character types but also structure types that contain character types.

To use saddr memory, it must be enabled; see --*data_model, -m*, page 200.

Note: If the saddr memory type is not enabled, the compiler can use the `EP` register and the special instructions for other purposes. For this reason, this feature should only be used when a small number of global or static variables will be accessed often (if speed is an issue) or exist in many locations (if you need to save code space).

USING DATA MEMORY ATTRIBUTES

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

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

Memory type	Keyword	Address range	Default in data model
Near	`__near`	±32 Kbytes around `0x0`	Tiny
Base-relative	`__brel`	64 Kbytes anywhere in RAM and 64 Kbytes anywhere in ROM	Small
Base-relative23	`__brel23`	8 Mbytes in RAM and 8 Mbytes in ROM. Only available for the V850E2M core and above.	Medium
Huge	`__huge`	Full memory	Large
Short addressing	`__saddr`	`EP` to `EP` + 256 bytes	—

Table 4: Memory types and their corresponding memory attributes

In this table, `GP` and `EP` are the Global Pointer and the Element Pointer which are aliases for the processor registers `R4` and `R30`, respectively. For more information, see *Memory access methods*, page 134.

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 206 for additional information.

For more information about each keyword, see *Descriptions of extended keywords*, page 243.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when variables are defined and in the declaration, see *General syntax rules for extended keywords*, page 239.

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

```
__near int i, j;
int __near k, l;
```

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

The `#pragma type_attribute` directive can also be used for specifying the memory attributes. The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Refer to the chapter *Pragma directives* for details about how to use the extended keywords together with pragma directives.

Type definitions

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

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

/* Defines directly */
__near char aByte;
/* No memory attribute necessary for pointers */
char __near *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 near memory.

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

__near struct MyStruct gamma;
```

This declaration is incorrect:

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

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer

to an integer in brel memory is declared. The function returns a pointer to an integer in huge memory. It makes no difference whether the memory attribute is placed before or after the data type.

`int MyA;`	A variable defined in default memory .
`int __brel MyB;`	A variable in brel memory.
`__huge int MyC;`	A variable in huge memory.
`int * MyD;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __brel * MyE;`	A pointer stored in default memory. The pointer points to an integer in brel memory.
`int __brel * __huge MyF;`	A pointer stored in huge memory pointing to an integer stored in brel memory.
`int __huge * MyFunction( int __brel *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in brel memory. The function returns a pointer to an integer stored in huge 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 155.

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

For more information about C++ classes, see *Using IAR attributes with Classes*, page 155.

Auto variables—on the stack

Variables that are defined inside a function—and not declared static—are named auto variables by the C standard. A few of these variables are placed in processor registers;

the rest are placed on the stack. From a semantic point of view, this is equivalent. The main differences are that accessing registers is faster, and that less memory is required compared to when variables are located on the stack.

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

THE STACK

The stack can contain:

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

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

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

Advantages

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

It is possible for a function to call itself 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`.

POTENTIAL PROBLEMS

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

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

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

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

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

Functions

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

Function-related extensions

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

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

Code models

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 function
- The maximum module size
- The maximum application size.

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

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

These code models are available:

Code model name	Description
Normal	Allows for up to 2 Mbytes of code memory
Large	No code memory limitation
Position-independent	Allows for up to 2 Mbytes of relocatable code memory

Table 5: Code models

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

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

Use the `--code_model` option to specify the code model for your project; see *--code_model*, page 199.

THE NORMAL CODE MODEL

The normal code model is the default code model. When this model is used, the natural assembler language instruction for performing function calls are used.

THE LARGE CODE MODEL

The large code model is designed to be used by applications that contain function calls that must reach more than 2 Mbytes. This could either be because the application itself is large, or because code is located in different parts of the memory at a greater distance than 2 Mbytes.

In this code model, a function call is more expensive than in the normal and position-independent code models.

THE POSITION-INDEPENDENT CODE MODEL

The position-independent code model is designed to be used in situations where the actual memory location of the code to be executed is not known at link time.

In this code model, plain function calls are not more expensive than in the normal code model. However, function calls via function pointers are much more expensive.

For more information about how to handle this type of code, see *Position-independent code*, page 64.

Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for V850 provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__interrupt`, `__callt`, `__syscall`,`__task`, `__trap`, and `__monitor`
- The pragma directive `#pragma vector`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, and `__set_interrupt_state`.

INTERRUPT FUNCTIONS

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

Interrupt service routines

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

The V850 microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the V850 microcontroller documentation from the chip manufacturer. If you want to handle several different interrupts using the same interrupt routine, you can specify several interrupt vectors.

Interrupt vectors and the interrupt vector table

For the V850 microcontroller, the interrupt vector is the offset into the interrupt vector table.

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime.

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

Defining an interrupt function—an example

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

```
#pragma vector = 0x40
__interrupt void MyInterruptRoutine(void)
{
  /* Do something */
}
```

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

System registers in interrupt functions

The processor has a collection of system registers and the type of interrupt controls which of them that are used. When an interrupt occurs, the program counter (`PC`) and processor status (`PSW`) are stored in specific system registers.

When generating code for interrupt functions, the compiler can add code to preserve some or all system registers. The interrupt vector controls whether and which registers are preserved, as shown:

Interrupt vector	Preserved registers for the V850 core	Preserved registers for the V850E core and above
`0x00`	None	None
`0x10-0x30`	`FE`*`xxx`*	`FE`*`xxx`*
`0x40-0x50`	`EI`*`xxx`*	`EI`*`xxx`*
`0x60`	`EI`*`xxx`*	`DB`*`xxx`*
`0x70-`	`EI`*`xxx`*	`EI`*`xxx`*

Table 6: Preserved registers in interrupt functions

An interrupt function specified without an interrupt vector will preserve all appropriate system registers.

On the V850E2M core and above, if `BSEL` (the bank select register) must be changed to access the appropriate registers, it too is preserved by the interrupt routine.

If the FPU is used on V850E2M or above, and the routine might change the floating-point status register, either directly or via a function call, the register is preserved by the interrupt routine.

Note that if the `__flat` function attribute is used, no system registers are preserved. In case you need to preserve and restore more interrupt registers, you can use the `__get_processor_register` and `__set_processor_register` intrinsic

functions. For more information, see *__flat*, page 244, *__get_processor_register*, page 276, and *__set_processor_register*, page 279.

Example

```
#include <intrinsics.h>

#pragma vector=0x40
__interrupt void my_interrupt(void)
{
  unsigned long saved_FEPC =
    __get_processor_register(Reg_CPU_FEPC);
  unsigned long saved_FEPSW =
    __get_processor_register(Reg_CPU_FEPSW);

  /* ... do something ... */

  __set_processor_register(Reg_CPU_FEPC, saved_FEPC);
  __set_processor_register(Reg_CPU_FEPSW, saved_FEPSW);
}
```

TRAP FUNCTIONS

A trap is a kind of exception that can be activated when a specific event occurs or is called, by using the processor instruction `TRAP`. In many respects, a trap function behaves as a normal function; it can accept parameters, and return a value.

The typical use for trap functions is for the client interface of an operating system. If this interface is implemented using trap functions, the operating system part of an application can be updated independently of the rest of the system.

Each trap function is typically associated with a vector. The header file `io`*`device`*`.h`, which corresponds to the selected device, contains predefined names for the existing exception vectors.

The `__trap` keyword and the `#pragma vector` directive can be used to define trap functions. For example, this piece of code defines a function doubling its argument:

```
/* No trap vector needed */
__flat __trap int Twice(int x)
{
  return x + x;
}
```

When a trap function is defined with a vector, the processor interrupt vector table is populated. It is also possible to define a trap function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime.

See the chip manufacturer's V850 microcontroller documentation for more information about the interrupt vector table.

When a trap function is used, the compiler ensures that the application also will include the appropriate trap-handling code. See the chapter *Assembler language interface* for more information.

When trap functions are being called using the processor instruction `TRAP`, the return address will point to the instruction itself. To return to the instruction after the `TRAP` instruction, the return address will by default be adjusted within the trap function.

CALLT FUNCTIONS

On the V850E microcontroller cores, the `CALLT` instruction can be used to call a fixed set of functions. The number of functions is limited to 64.

This type of function is intended to be used in roughly the same situations as trap functions. The `CALLT` instruction only exists for the V850E microcontroller cores and above.

The advantage over `TRAP` functions is that a system can contain 64 callt functions, whereas only 32 trap functions can be defined. It is also more efficient to call a callt function.

Each callt function must be associated with a vector ranging from 0 to 63. The `__callt` keyword and the `#pragma vector` directive can be used to define callt functions. For example, the following piece of code defines a function doubling its argument:

```
#pragma vector=15
__flat __callt int twice(int x)
{
  return x + x;
}
```

SYSCALL FUNCTIONS

On the V850E2M microcontroller core, the `SYSCALL` instruction can be used to call a fixed set of functions. The number of functions is limited to 256.

This type of function is intended to be used in roughly the same situations as trap functions. The `SYSCALL` instruction only exists for the V850E2M microcontroller core and above.

The advantage over `TRAP` functions is that a system can contain 256 syscall functions, whereas only 32 trap functions can be defined. It is also more efficient to call a syscall function.

This instruction is dedicated to calling the system service of an operating system.

Each syscall function must be associated with a vector ranging from 0 to 255. The `__syscall` keyword and the `#pragma vector` directive can be used to define syscall functions. For example, the following piece of code defines a function doubling its argument:

```
#pragma vector=15
__flat __syscall int twice(int x)
{
  return x + x;
}
```

MONITOR FUNCTIONS

A monitor function causes interrupts to be disabled during execution of the function. At function entry, the status register is saved and interrupts are disabled. At function exit, the original status register is restored, and thereby the interrupt status that existed before the function call is also restored.

To define a monitor function, you can use the `__monitor` keyword. For more information, see *__monitor*, page 246.

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

Example of implementing a semaphore in C

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

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


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

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

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

    return 0;
  }
}


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

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


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

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

```
/* An example of using the semaphore. */

void MyProgram(void)
{
  GetLock();

  /* Do something here. */

  ReleaseLock();
}
```

Example of implementing a semaphore in C++

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

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

```
#include <intrinsics.h>

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

    // Disable all interrupts.
    __disable_interrupt();
  }

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

private:
  __istate_t mState;
};
```

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

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

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

private:
  static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
  static long nextStop = 100;

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

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, two restrictions apply:

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

- Callt and trap member functions cannot be declared virtual. The reason for this is that callt and trap functions cannot be called via function pointers.

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 258.

- `--use_c++_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.

- `--aggressive_inlining` makes the compiler inline more functions, see *--aggressive_inlining*, page 197.

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 171.

For more information about the function inlining optimization, see *Function inlining*, page 174.

Position-independent code

Position-independent code is designed to be used when the physical address of the application is not known at link time. A typical example of this is an application loaded into an embedded system by an operating system at runtime.

When building an application using the position-independent code model, the linker will treat it as a normal, absolute-located application. However, the code generated by the compiler does not assume that it will be placed and executed at any location in memory.

THE DISTANCE MOVED

If the application needs to know where it is executing, it can use the intrinsic function `__code_distance` to get the difference between the absolute location it was linked for and the location where it executes. See *__code_distance*, page 273.

CALLING FUNCTIONS OUTSIDE THE APPLICATION

An embedded application might in some situations call functions that are located outside the application itself. This could for example be calls to the operating system or some kind of on-chip ROM-monitor. A call of this kind can be performed using one of the special function types trap or callt, assuming that the trap and callt vectors have been properly initialized by the operating system. Another method would be to call a function via function pointers, if the location of the function is known.

In the position-independent code model, a function pointer is assumed to contain the address that the linker assigned the function. When the function call is performed, the distance the code has been moved is compensated for.

To assign a function pointer the value of a physical address, the function pointer must first be converted to a position-independent function pointer. You can do this using the intrinsic function `__absolute_to_pic`.

Example

Here a function located at the physical address `0x10000` is called:

```
#include <intrinsics.h>

typedef void (fp_t)(void);

void call_the_operating_system(void)
{
  fp_t * pointer = (fp_t *)__absolute_to_pic(0x10000);
  (*pointer)();
}
```

Likewise, it is possible to convert the pointer of a function in the position-independent code to a pointer to a physical address using the `__pic_to_absolute` intrinsic function. This could be useful if a function pointer should be passed outside the application to, for instance, an operating system.

Placing code and data

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

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

Segments and memory

In an embedded system, there 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 a separate 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 has 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.

By default, the compiler uses these XLINK segment memory types:

Segment memory type	Description
`CODE`	For executable code
`CONST`	For data placed in ROM
`DATA`	For data placed in RAM

Table 7: XLINK segment memory types

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

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

Placing segments in memory

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

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. For showing the methods, fictitious examples are used.

CUSTOMIZING THE LINKER CONFIGURATION FILE

The `config` directory contains ready-made linker configuration files for all supported devices (filename extension `xcl`). The files contain the information required by the

linker, and are ready to be used. The only change you will normally have to make to the supplied linker configuration file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you must add details about the external RAM memory area.

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

Range	Type
`0x00000–0x1FFFF`	ROM
`0x20000–0x3FFFF`	RAM
`0xFFFF8000–0xFFFFEFFF`	RAM

Table 8: Memory layout of a target system (example)

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

Do not modify the original file. We recommend that you make a copy in the working directory, and modify and use the copy instead.

The contents of the linker configuration file

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

- The CPU used:

  ```
  -cv850
  ```

  This specifies your target microcontroller.

  Note that the parameter should be `v850` for all members of the V850 microcontroller family.
- Definitions of constants used in the file. These are defined using the XLINK option `-D`.
- The placement directives (the largest part of the linker configuration file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

In the linker configuration file, 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.

See the *IAR Linker and Library Tools Reference Guide* for more information.

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 `0x1000-0x1FFF`.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=1000-1FFF
```

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

```
-Z(CONST)MYSEGMENTA=1000-1FFF
-Z(CODE)MYCODE
```

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

```
-Z(CONST)MYSMALLSEGMENT=1000-10FF
-Z(CONST)MYLARGESEGMENT=1000-1FFF
```

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=20000-21FFF,FFFF0000-FFFF1FFF
```

If your application has an additional RAM area in the memory range `0x3F000-0x3F7FF`, you can simply add that to the original definition:

```
-P(DATA)MYDATA=20000-21FFF,0x3F000-0x3F7FF,FFFF0000-FFFF1FFF
```

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.

Data segments

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

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different data models available in the compiler. For information about these details, see the chapter *Data storage*.

STATIC MEMORY SEGMENTS

Static memory is memory that contains variables that are global or declared static, see the chapter *Data storage*. Variables declared static can be divided into these categories:

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

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

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

Segment naming

The names of the segments consist of two parts—the segment group name and a *suffix*—for instance, `BREL_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 BREL and __brel. The following table summarizes the memory types and the corresponding segment groups:

Memory type	Segment group	Memory range
Near	NEAR	0x00000000-0x00007FFF 0xFFFF8000-0xFFFFFFFF
Base-relative	BREL	Anywhere
Base-relative23	BREL23	Anywhere
Huge	HUGE	Anywhere
Short addressing with a 128-byte offset	SADDR7	Anywhere
Short addressing with a 256-byte offset	SADDR8	Anywhere

Table 9: Memory types with corresponding segment groups

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more information about segment memory types, see *Segment memory type*, page 68.

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	Suffix	Segment memory type
Start placeholder*	BASE	varies
Non-initialized data	N	DATA
Zero-initialized data	Z	DATA
Non-zero initialized data	I	DATA
Initializers for the above	ID	CONST
Constants†	C	CONST
Non-initialized absolute addressed data	AN	DATA
Constant absolute addressed data	AC	CONST

Table 10: Segment name suffixes

* There are only three start placeholders: BREL_BASE, BREL_CBASE, and SADDR_BASE.

† Constants placed in saddr memory will be stored in RAM, that is in segments with the suffix Z or I.

For information about all supported segments, see *Summary of segments*, page 299.

Examples

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

`__near int j;` `__near int i = 0;`	The near variables that are to be initialized to zero when the system starts are placed in the segment `NEAR_Z`.
`__no_init __near int j;`	The near non-initialized variables are placed in the segment `NEAR_N`.
`__near int j = 4;`	The near non-zero initialized variables are placed in the segment `NEAR_I` in RAM, and the corresponding initializer data in the segment `NEAR_ID` in ROM.

Initialized data

When an application is started, the system startup code initializes static and global variables in these steps:

1. It clears the memory of the variables that should be initialized to zero.
2. It initializes the non-zero variables by copying a block of ROM to the location of the variables in RAM. This means that the data in the ROM segment with the suffix `ID` is copied to the corresponding `I` segment.

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

   - The other segment is divided in exactly the same way
   - It is legal to read and write the memory that represents the gaps in the sequence.

   For example, if the segments are assigned these ranges, the copy might fail:

`BREL_I`	`0x1000-0x10FF` and `0x1200-0x12FF`
`BREL_ID`	`0x4000-0x41FF`

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

`BREL_I`	`0x1000-0x10FF` and `0x1200-0x12FF`
`BREL_ID`	`0x4000-0x40FF` and `0x4200-0x42FF`

   The `ID` segment can, for all segment groups, be placed anywhere in memory, because it is not accessed using the corresponding access method. Note that the gap between the ranges will also be copied.

3 Finally, global C++ objects are constructed, if any.

Data segments for static memory in the default linker configuration file

The default linker configuration file contains these directives to place the static data segments:

```
/* First, the segments to be placed in ROM are defined. */
-Z(CONST)NEAR_C=0000-7FFF
-Z(CONST)SADDR7_ID,SADDR8_ID,NEAR_ID,BREL_ID,HUGE_ID,
         HUGE_C=0000-1FFFF
-Z(CONST)BREL_CBASE,BREL_C, BREL23_C

/* Then, the RAM data segments are placed in memory. */
-Z(DATA)BREL_BASE,BREL_I,BREL_Z,BREL_N,BREL23_I,BREL23_Z,
        BREL23_N=20000-2FFFF
-Z(DATA)SADDR_BASE=3FF00
-Z(DATA)SADDR7_I,SADDR7_Z,SADDR7_N=3FF00-3FF7F
-Z(DATA)SADDR8_I,SADDR8_Z,SADDR8_N=3FF00-3FFFF
-Z(DATA)HUGE_I,HUGE_Z,HUGE_N=20000-3FFFF
-Z(DATA)NEAR_Z,NEAR_I,NEAR_N=FFFF8000-FFFFEFFF
```

Note that the `SADDR7` and `SADDR8` segment groups share parts of the same memory area. This allows `SADDR8` segments to be placed in memory not used by the `SADDR7` segment group.

THE STACK

The stack is used by functions to store variables and other information that is used locally by functions, see the chapter *Data storage*. It is a continuous block of memory pointed to by the processor stack pointer register `SP`.

The data segment used for holding the stack is called `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 done differently using the command line interface as compared to when using the IDE.

Stack size allocation in the IDE

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

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

Stack size allocation from the command line

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

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

```
-D_CSTACK_SIZE=size
```

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

Specify an appropriate size for your application. Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

Placement of stack segment

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

```
-Z(DATA)CSTACK+_CSTACK_SIZE=FFFF8000-FFFFEFFF
```

Note: This range does not specify the size of the stack; it specifies the range of the available memory

Stack size considerations

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

THE HEAP

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

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

- The linker segment used for the heap
- The steps involved 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.

The memory allocated to the heap is placed in the segment `HEAP`, which is only included in the application if dynamic memory allocation is actually used.

Heap size allocation in the IDE

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

Add the required heap size in the **Heap size** text box.

Heap size allocation from the command line

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

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

```
-D_HEAP_SIZE=size
```

Normally, this line is prefixed with the comment character `//` because the IDE controls the heap size allocation. To make the directive take effect, remove the comment character.

Specify the appropriate size for your application. If you use a heap, you must allocate at least 50 bytes for it. Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

Placement of heap segment

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

```
-Z(DATA)HEAP+_HEAP_SIZE=FFFF8000-FFFFEFFF
```

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

Heap size and standard I/O

If your DLIB runtime environment is configured to use `FILE` descriptors, as in the Full configuration, input and output buffers for file handling will be allocated. In that case, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an V850 microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

LOCATED DATA

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in a segment called `GLOBAL_AN`. The

individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker configuration file.

USER-DEFINED SEGMENTS

If you create your own segments 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.

Code segments

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 299.

Pay attention to the limitation of the placement of the different segments. One limitation that is often mentioned is that code should be callable using the normal call or branch assembler instructions. These are limited to calls where the destination is located no more than 2 Mbytes from the calling function.

STARTUP CODE

The segment `CSTART` contains code used during system startup (`cstartup`) and system termination (). The system startup code should be placed within 2 Mbytes from the location where the chip starts executing code after a reset. For the V850 microcontroller, this is at the address `0x0`. The segments must also be placed into one continuous memory space, which means that the `-P` segment directive cannot be used.

In the default linker configuration file, this line states that this segment can be placed anywhere in the `0x0-0x1FFFF` range:

```
-Z(CODE)CSTART=00000-1FFFF
```

NORMAL CODE

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

```
-Z(CODE)CODE=0000-1FFF
```

INTERRUPT VECTORS AND FUNCTIONS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the segment `INTVEC`. You must place this segment at the address `0x0`. The linker directive would then look like this:

```
-Z(CONST)INTVEC=0
```

Interrupt functions are placed in the ICODE segment. This segment must be located so that it can be called using normal instructions (with a 2 Mbyte range) from the INTVEC segment. The linker directive will look like this:

```
-Z(CODE)ICODE=0000-1FFFF
```

TRAP VECTORS

The trap vector table is located in its own segment, TRAPVEC. The TRAPVEC segment can be placed using the linker command directive:

```
-Z(CONST)TRAPVEC=0000-1FFFF
```

CALLT FUNCTIONS

Callt functions are placed in the CLTCODE segment.

When a callt function is defined and has a vector, an entry in the callt vector table is generated. The table is placed in the CLTVEC segment.

The CLTCODE segment must be located within 64 Kbytes of the CLTVEC segment. In the following linker command directive example, the CLTCODE segment will be placed immediately after the CLTVEC segment.

```
-Z(CONST)CLTVEC=0000-1FFFF
-Z(CODE)CLTCODE
```

Note: Callt functions are available for the V850E core and above.

SYSCALL FUNCTIONS

Syscall functions are placed in the SYSCALLCODE segment.

When a syscall function is defined and has a vector, an entry in the syscall vector table is generated. The table is placed in the SYSCALLVEC segment.

In the following linker command directive example, the SYSCALLCODE segment will be placed immediately after the SYSCALLVEC segment.

```
-Z(CONST)SYSCALLVEC=0000-1FFFF
-Z(CODE)SYSCALLCODE
```

Note: Syscall functions are only available for the V850E2M core and above.

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=0000-1FFFF
```

`DIFUNCT` must be placed using `-Z`. For additional information, see *DIFUNCT*, page 307.

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

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

Normally, XLINK will not generate an output file if any errors, such as range errors, occur during the linking process. Use the option **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*.

The DLIB runtime environment

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

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 `v850\lib` and `v850\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 arithmetic support modules like floating-point coprocessors.
- 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 295.
- 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 a prebuilt library*, page 83.
- 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 100.
- 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 87. You can also specify the size and placement of the stack and the heap, see *The stack*, page 74, and *The heap*, page 75, 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 89.
- 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 92.

- 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 93.

- 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 95 and *Customizing system initialization*, page 99.

- 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 94.

- Check module consistency

  You can use runtime model attributes to ensure that modules are built using compatible settings, see *Checking module consistency*, page 112.

Using a prebuilt library

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

- Processor variant
- Library configuration—Normal or Full
- Code model
- Floating-point unit.

Note that all prebuilt runtime libraries are built using the Small data model. However, they can be used by an application built using any data model.

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 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, for instance:

  `dl85nn1.r85`

- 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:\...\dl85nf1.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 `v850\lib\`.

These prebuilt runtime libraries are available:

Library	Code model	Library configuration	CPU variant	FPU
dl85nn0.r85	Normal	Normal	0	
dl85ln0.r85	Large	Normal	0	
dl85pn0.r85	Position-independent	Normal	0	
dl85nf0.r85	Normal	Full	0	
dl85lf0.r85	Large	Full	0	
dl85pf0.r85	Position-independent	Full	0	
dl85nn1.r85	Normal	Normal	1	
dl85nn1fpu.r85	Normal	Normal	1	Single
dl85ln1.r85	Large	Normal	1	
dl85ln1fpu.r85	Large	Normal	1	Single
dl85pn1.r85	Position-independent	Normal	1	
dl85pn1fpu.r85	Position-independent	Normal	1	Single
dl85nf1.r85	Normal	Full	1	
dl85nf1fpu.r85	Normal	Full	1	Single
dl85lf1.r85	Large	Full	1	
dl85lf1fpu.r85	Large	Full	1	Single
dl85pf1.r85	Position-independent	Full	1	
dl85pf1fpu.r85	Position-independent	Full	1	Single
dl85nn3.r85	Normal	Normal	3	

Table 11: Prebuilt libraries

Library	Code model	Library configuration	CPU variant	FPU
dl85nn3fpu32.r85	Normal	Normal	3	Single
dl85nn3fpu64.r85	Normal	Normal	3	Double
dl85ln3.r85	Large	Normal	3	
dl85ln3fpu32.r85	Large	Normal	3	Single
dl85ln3fpu64.r85	Large	Normal	3	Double
dl85pn3.r85	Position-independent	Normal	3	
dl85pn3fpu32.r85	Position-independent	Normal	3	Single
dl85pn3fpu64.r85	Position-independent	Normal	3	Double
dl85nf3.r85	Normal	Full	3	
dl85nf3fpu32.r85	Normal	Full	3	Single
dl85nf3fpu64.r85	Normal	Full	3	Double
dl85lf3.r85	Large	Full	3	
dl85lf3fpu32.r85	Large	Full	3	Single
dl85lf3fpu64.r85	Large	Full	3	Double
dl85pf3.r85	Position-independent	Full	3	
dl85pf3fpu32.r85	Position-independent	Full	3	Single
dl85pf3fpu64.r85	Position-independent	Full	3	Double

Table 11: Prebuilt libraries (Continued)

Library filename syntax

The names of the libraries are constructed from these elements:

{*library*}	is `dl` for the IAR DLIB runtime environment.
{*cpu*}	is `85`.
{*code_model*}	is one of `n`, `l`, or `p` for the Normal, Large or Position-independent code model, respectively.
{*lib_config*}	is one of `n` or `f` for normal and full, respectively.
{*cpu_variant*}	is either `0` for the V850, `1` for the V850E and V850E2, or `3` for the V850E2M core.
{*fpu*}	For processor cores with only a single-precision FPU, {*fpu*} is `fpu`. For processor cores that support both single-precision and double-precision FPUs, {*fpu*} is `fpu32` for the single precision, or `fpu64` for the double precision, respectively.

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

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

These items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 87
Startup and termination code	*System startup and termination*, page 95
Low-level input and output	*Standard streams for input and output*, page 101
File input and output	*File input and output*, page 105
Low-level environment functions	*Environment interaction*, page 108
Low-level signal functions	*Signal and raise*, page 109
Low-level time functions	*Time*, page 109
Some library math functions	*Math functions*, page 110
Size of heaps, stacks, and segments	*Placing code and data*, page 67

Table 12: Customizable items

For information about how to override library modules, see *Overriding library modules*, page 93.

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 103.

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

Table 13: Formatters for printf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 103.

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 `c`, `d`, `i`, `o`, `p`, `s`, `u`, `X`, `x`, and `%`	Yes	Yes	Yes
Multibyte support	Yes/No	Yes/No	Yes/No
Floating-point specifiers `a`, and `A`	No	No	Yes
Floating-point specifiers `e`, `E`, `f`, `F`, `g`, and `G`	No	No	Yes
Conversion specifier `n`	No	No	Yes
Scan set `[` and `]`	No	Yes	Yes
Assignment suppressing `*`	No	Yes	Yes
`long long` support	No	No	Yes

Table 14: Formatters for scanf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 103.

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 debugger 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	**Debug information for C-SPY**	`-Fubrof`	Debug support for C-SPY without any runtime support
Runtime debugging*	**With runtime control modules**	`-r`	The same as `-Fubrof`, but also includes debugger support for handling program abort, exit, and assertions.

Table 15: Levels of debugging support in runtime libraries

Debugging support	Linker option in the IDE	Linker command line option	Description
I/O debugging*	**With I/O emulation modules**	`-rt`	The same as `-r`, but also includes debugger support for I/O handling, which means that `stdin` and `stdout` are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.

Table 15: Levels of debugging support in runtime libraries (Continued)

* 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 90.

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 V850*.

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

Table 16: 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

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 93.

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 90.

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 `v850\src\lib` directory.

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

- *Standard streams for input and output*, page 101
- *File input and output*, page 105
- *Signal and raise*, page 109
- *Time*, page 109
- *Assert*, page 112.

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 92. 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: The code model, include paths, and the library configuration file must be the same for the library module as for the rest of your code. The include path should also include the library source directory.

Some library files must be built using the same data model as the runtime library (the prebuilt libraries use the Small data model). In that case, also use the command line option `__no_data_model_rt_attribute`, see *--no_data_model_rt_attribute*, page 215. The IDE does not provide a method for building an application containing modules using different data models.

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.

The library files that you can override with your own versions are located in the `v850\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 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*.

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 17, *Library configurations*.

In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 37.

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 and documented in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities. Each library also has its own library configuration header file, which sets up that specific library's configuration. For more information, see Table 12, *Customizable items*.

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 library configuration file 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.

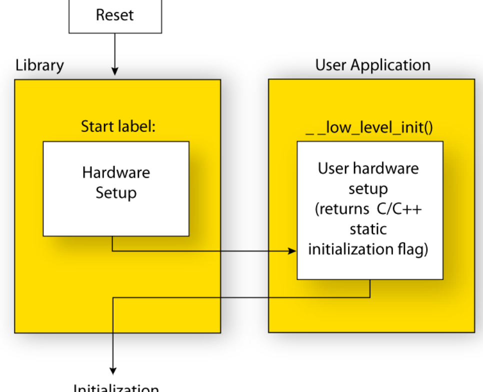
The code for handling startup and termination is located in the source files `cstartup.s85`, `cexit.s85`, and `low_level_init.c` located in the `v850\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 99.

SYSTEM STARTUP

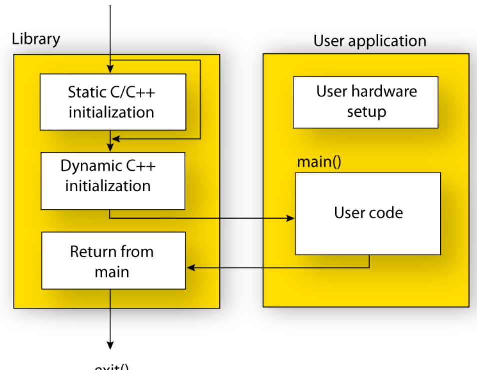
During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

For the hardware initialization, it looks like this:



- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code.
- The stack pointer (`SP`) is initialized
- The brel RAM pointer (`GP`), and the brel ROM base pointer (`R25`), are initialized to point to the base-relative memory areas. In fact, they point to a location 32 Kbytes from the beginning, as described in the chapter *Assembler language interface*
- If register constants are used, the registers `R18` and `R19` are set to `255` and `65535`, respectively
- If short addressing is enabled, the corresponding base register—`EP`—is initialized
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

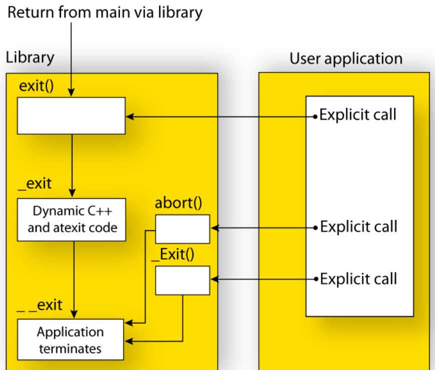
For the C/C++ initialization, it looks like this:



- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialized data*, page 73
- When the position-independent code model is used, the distance the code was moved is calculated and stored in the variable `?CODE_DISTANCE`
- The callt system registers are initialized, if needed.
- The syscall system registers are initialized, if used.
- The `CTBP` special system register is initialized to the beginning of the callt vector table, if callt functions are used in the application
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

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



An application can terminate normally in two different ways:

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

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

The default `exit` function is written in C. It calls a small assembler function `_exit` that will 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 89.

Customizing system initialization

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

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cstartup` before the data segments are initialized. Modifying the file `cstartup.s85` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s85` and `low_level_init.c`, located in the `v850\src\lib` directory.

Note: Normally, you do not need to customize either of the files `cstartup.s85` or `cexit.s85`.

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 94.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s85`, you do not have to rebuild the library.

__LOW_LEVEL_INIT

A skeleton low-level initialization file is supplied with the product: `low_level_init.c`. Note that static initialized variables cannot be used within the file, because variable initialization has not been performed at this point.

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

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

MODIFYING THE FILE CSTARTUP.S85

As noted earlier, you should not modify the file `cstartup.s85` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify

the file `cstartup.s85`, 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 93.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s85`. 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 `printf` and `scanf`, and no hexadecimal floating-point numbers in `strtod`.
Full DLIB	Full locale interface, C locale, file descriptor support, multibyte characters in `printf` and `scanf`, and hexadecimal floating-point numbers in `strtod`.

Table 17: 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 205.

- 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 94.

The prebuilt libraries are based on the default configurations, see Table 17, *Library configurations*.

Standard streams for input and output

Standard text input and output 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 92.

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 `v850\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 94. 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 89.

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 `0xFFFFF308`:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ FFFFF308;

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 `0xFFFF8000`:

```
#include <stddef.h>

__no_init __near volatile unsigned char kbIO @ 0xFFFF8000;

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 167.

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 87.

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

Table 18: 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
_DLIB_SCANF_MULTIBYTE	Multibyte characters
_DLIB_SCANF_LONG_LONG	Long long (ll qualifier)
_DLIB_SCANF_SPECIFIER_FLOAT	Floating-point numbers
_DLIB_SCANF_SPECIFIER_N	Output count (%n)
_DLIB_SCANF_QUALIFIERS	Qualifiers h, j, l, t, z, and L
_DLIB_SCANF_SCANSET	Scanset ([*])
_DLIB_SCANF_WIDTH	Width
_DLIB_SCANF_ASSIGNMENT_SUPPRESSING	Assignment suppressing ([*])

Table 19: 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 94.
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 92.

Note that file I/O capability in the library is only supported by libraries with the full library configuration, see *Library configurations*, page 100. 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
`__close`	`close.c`	Closes a file.
`__lseek`	`lseek.c`	Sets the file position indicator.
`__open`	`open.c`	Opens a file.
`__read`	`read.c`	Reads a character buffer.
`__write`	`write.c`	Writes a character buffer.
`remove`	`remove.c`	Removes a file.
`rename`	`rename.c`	Renames a file.

Table 20: 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 89.

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 94.

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 `v850\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 93.

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 94.

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 89.

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 v850\src\lib directory. For information about overriding default library modules, see *Overriding library modules*, page 93.

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 94.

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 296.

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 v850\src\lib directory. For information about overriding default library modules, see *Overriding library modules*, page 93.

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 94.

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 89.

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 94.

Math functions

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

SMALLER VERSIONS

The functions `cos`, `exp`, `log`, `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_log10_small=log10
-e__iar_exp_small=exp
-e__iar_pow_small=pow
-e__iar_Sin_small=__iar_Sin
```

```
-e__iar_sin_smallf=sinf
-e__iar_cos_smallf=cosf
-e__iar_tan_smallf=tanf
-e__iar_log_smallf=logf
-e__iar_log10_smallf=log10f
-e__iar_exp_smallf=expf
-e__iar_pow_smallf=powf
-e__iar_Sin_smallf=__iar_Sinf

-e__iar_sin_smalll=sinl
-e__iar_cos_smalll=cosl
-e__iar_tan_smalll=tanl
-e__iar_log_smalll=logl
-e__iar_log10_smalll=log10l
-e__iar_exp_smalll=expl
-e__iar_pow_smalll=powl
-e__iar_Sin_smalll=__iar_Sinl
```

Note that if `cos` or `sin` is redirected, `__iar_Sin` must be redirected as well.

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 `cos` or `sin` is redirected, `__iar_Sin` must be redirected as well. The same applies to `pow` and `__iar_Pow`.

Assert

If you linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `v850\src\lib` directory. For more information, see *Building and using a customized library*, page 94. 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 286.

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 lock two or more registers. If you write a module that assumes that these registers are locked, it is possible to check that the module is not used in an application that must be able to write to all registers.

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

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

Table 21: Example of runtime model attributes

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

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

USING RUNTIME MODEL ATTRIBUTES

To ensure module consistency with other object files, use the `#pragma rtmodel` directive to specify runtime model attributes in your C/C++ source code. For example, 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 264 and the *IAR Assembler Reference Guide for V850*, 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_model`	`normal`, `large`, or `pic`	Corresponds to the `--code_model` option used for the project.
`__cpu`	`v850` or `v850e`	Corresponds to the core used for the project, either V850 or a higher core. This key can be used to avoid linking code designed for one processor core with an application designed for another one.
`__fpu_double`	`v850e2`	This key is defined when double precision FPU is enabled.
`__fpu_single`	`v850e1`, `v850e2`, or undefined	This key is defined when the FPU is enabled. Its value corresponds to the FPU variant used.
`__memory_model`	`tiny`, `small`, `medium`, or `large`	Corresponds to the `--data_model` option used for the project.
`__reg_ep`	`frame`, `saddr`, or undefined	This key has the value `frame` in data models without short addressing. If a saddr variable is used in data models with short addressing, this key has the value `saddr`. If no saddr variables are used in a data model with short addressing the value is undefined, making it possible to link such a module with modules both with and without short addressing.

Table 22: Predefined runtime model attributes

Runtime model attribute	Value	Description
__reg_lock2	free, const, locked, or *	This key has the value free when the registers R18 and R19 are not locked. If they are locked, and you are using the --reg_const option, the value is const. If they are locked, and you are using the option --lock_regs_compatibility, the value is *, making it possible to link such a module with modules with no registers locked. If they are locked, and you are not using any of these options, the value is locked.
__reg_lock6	free, locked, or *	This key has the value free when the registers R17 and R20–R22 are not locked. If they are locked, and you are using the option --lock_regs_compatibility, the value is *, making it possible to link such a module with modules with fewer registers locked. If they are locked, and you are not using this option, the value is locked.
__reg_lock10	free, locked, or *	This key has the value free when the registers R15–R16 and R23–R24 are not locked. If they are locked, and you are using the option --lock_regs_compatibility, the value is *, making it possible to link such a module with modules with fewer registers locked. If they are locked, and you are not using this option, the value is locked.
__reg_r25	brel_const	This key is provided for future compatibility. It has no current use.
__rt_version	9	This runtime key is always present in all modules generated by the compiler. If a major change in the runtime characteristics occurs, the value of this key changes.

Table 22: Predefined runtime model attributes (Continued)

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 Reference Guide for V850.*

Note: In addition to these attributes, compatibility is also checked against the AEABI runtime attributes. These attributes deal mainly with what device to use, etc, and they are not user-configurable.

Example

The following assembler source code provides a function that increases the register R6 to count the number of times it was called. The routine assumes that the application does not use R6 for anything else, that is, the register is locked for usage. To ensure this, a runtime module attribute, __reg_r6, is defined with a value counter. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not set this attribute. Note that the compiler sets this attribute to free, unless the register is locked.

```
            module    myCounter
            public    myCounter
            rseg      CODE:CODE(2)
            code
            rtmodel   "__reg_r6", "counter"
myCounter:  add       1,r6
            jmp       [lp]
            end
```

If this module is used in an application that contains modules where the register R6 is not locked, the linker issues an error:

```
Error[e117]: Incompatible runtime models. Module myCounter
specifies that '__reg_r6' must be 'counter', but module part1
has the value 'free'
```

Assembler language interface

When you develop an application for an embedded system, there might be situations where you will find it necessary to write parts of the code in assembler, for example when using mechanisms in the V850 microcontroller that require precise timing and special instruction sequences.

This chapter describes the available methods for this and some C alternatives, with their advantages and disadvantages. It also describes how to write functions in assembler language that work together with an application written in C or C++.

Finally, the chapter covers how functions are called in the different code models, the different memory access methods corresponding to the supported memory types, and how you can implement support for call frame information in your assembler routines for use in the C-SPY® Call Stack window.

Mixing C and assembler

The IAR C/C++ Compiler for V850 provides several ways to access low-level resources:

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

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

INTRINSIC FUNCTIONS

The compiler provides a few predefined functions that allow direct access to low-level processor operations without having to use the assembler language. These functions are known as intrinsic functions. They can be very useful in, for example, time-critical routines.

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

This causes some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all scratch registers are destroyed by an inline assembler instruction. In many cases, the overhead of the extra instructions is compensated for by the work of the optimizer.

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

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

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

The first question is discussed in the section *Calling assembler routines from C*, page 121. The following two are covered in the section *Calling convention*, page 124.

For information about how data in memory is accessed, see *Memory access methods*, page 134.

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 139.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 121, and *Calling assembler routines from C++*, page 123, respectively.

INLINE ASSEMBLER

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

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

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

The syntax is:

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

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

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

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

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

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

static __near char sFlag;

void Foo(void)
{
  while (!sFlag)
  {
    asm("LD.B        UART1_SR[r0],r1\n"
        "ST.B        r1,sFlag[r0]");
  }
}
```

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

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

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

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Alignment cannot be controlled; this means, for example, that `DC32` directives might be misaligned
- Auto variables cannot be accessed.

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

Calling assembler routines from C

An assembler routine that will be called from C must:

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

  ```
  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 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:

```
iccv850 skeleton.c -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s85`. Also remember to specify the code model and data model you are using, a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s85`.

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 139.

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.

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++:

```
#ifdef __cplusplus
extern "C"
{
#endif

int F(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general V850 CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

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

Any of the registers `R1` and `R5-R9`, and any other register that is used as a parameter register, can be used as a scratch register by the function.

Special function types have no scratch registers, except the parameter registers.

Preserved registers

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

The registers `R10` through to `R30`, except the registers that are used as parameter registers, are preserved registers.

For information about system registers in interrupt functions, see *System registers in interrupt functions*, page 56.

Special registers

For some registers, you must consider certain prerequisites:

- `R0` will act as a normal processor register with the exception that the value of the register is always zero
- Some registers might possibly be unavailable due to register locking. For more information, see --*lock_regs*, page 211.
- The stack pointer register must at all times point to the topmost element on the stack. In the eventuality of an interrupt, everything on the other side of the point the stack pointer points to, can be destroyed.
- The brel base pointer registers `GP` and `R25` (pointing to data areas that are addressed with indexed addressing modes) must never be changed.
- The link register holds the return address at the entrance of the function.

FUNCTION ENTRANCE

Parameters can be passed to a function using one of two basic methods: in registers or on the stack. It is much more efficient to use registers than to take a detour via memory, so the calling convention is designed to use registers as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. The parameters are also passed on the stack in these cases:

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

Note: Interrupt functions cannot take any parameters.

Hidden parameters

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

- If the function returns a structure, the memory location where the structure will be stored is passed as an extra parameter. Notice that it is always treated as the first parameter.
- If the function is a non-static C++ member function, then the `this` pointer is passed as the first parameter (but placed after the return structure pointer, if there is one). The reason why the member function must be non-static is that static member methods do not have a `this` pointer.

Register parameters

Scalar parameters—integers and pointers—require one register. The same is true for `float` values. On the other hand, `double` and `long long` values require two registers.

The registers available for passing parameters are `R1` and `R5–R19`.

Parameters	Passed in registers
8-, 16-, or 32-bit values	`R1`, `R5–R19`
64-bit values	`(R5:R1)`, `(R7:R6)`, `(R9:R8)`, `(R11:R10)`, `(R13:R12)`, `(R15:R14)`, `(R17:R16)`, `(R19:R18)`

Table 23: Registers used for passing parameters

Note that:

1 The register `R9` is not available for the V850 core, because it is used as a scratch register. When the Large code model is used, `R9` is unavailable for all cores.

2 `R15–R19` are possibly not available because of register locking.

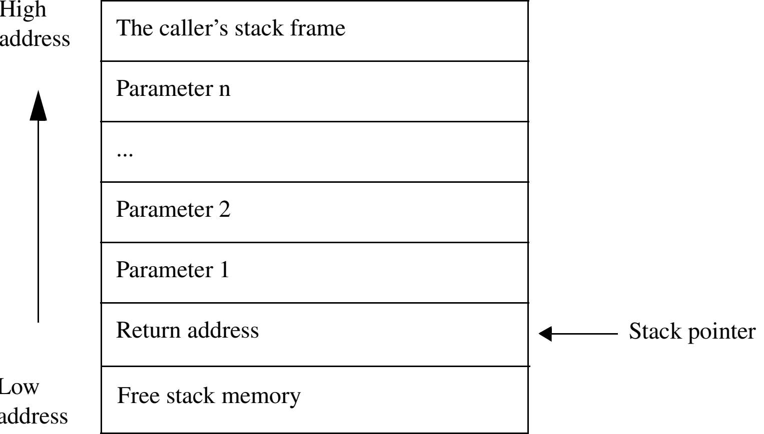
The assignment of registers to parameters is a straightforward process. The first parameter is assigned to the first available register, the second parameter to the second available register etc. Should there be no more available registers, the parameter is passed on the stack.

If a `double` parameter is passed in registers, it can only be passed using a register pair listed in the table. If, for example, `R1` has been assigned to a scalar parameter, the next available register pair is `R7:R6`. The register `R5` will be allocated by the next scalar parameter, if any.

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 4, 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

Scalar and `float` values are returned using register `R1`. `double` and `long long` values use the register pair `R5:R1`.

If a structure is returned, the caller of the function is responsible for allocating memory for the return value. A pointer to the memory is passed as a “hidden” first parameter that is always allocated to register `R1`. The called function must return the value of the location in register `R1`.

Stack layout at function exit

It is the responsibility of the caller to clean the stack after the called function returns.

Return address handling

A function written in assembler language should, when finished, return to the caller. At a function call, the return address is stored in the return address register `LP`.

Typically, a function returns by using the `JMP` instruction, for example:

```
jmp     [LP]
```

If a function is to call another function, the original return address must be stored somewhere. This is normally done on the stack. This example shows how it can be done for the cores V850E and above:

```
name    call
rseg    CODE:CODE(2)
extern  _func
code

prepare     {lp},0
jarl        _func,lp

; Do something here.

dispose     0,{lp},[lp]
```

This is the equivalent example for the V850 core:

```
add         -4,sp
st.w        lp,0[sp]
jarl        _func,lp

; Do something here.

ld.w        0[sp],lp
add         4,sp
jmp         [lp]

end
```

For the V850 core, the return address is restored directly from the stack with the `LD` instruction. For the V850E core and above, the `DISPOSE` instruction is used.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

These restrictions apply to the special function types `__callt`, `__interrupt`, `__syscall`, and `__trap`:

- The return address is not stored in the register `LP` but in dedicated system registers.
- The special function types have no scratch registers except for the parameter registers.

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 add_one(int);
```

This function takes one parameter in the register `R1`, and the return value is passed back to its caller in the register `R1`.

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

```
            name    return
            rseg    CODE:CODE(2)
            code
            add     1,r1
            jmp     [lp]

            end
```

Example 2

This example shows how structures are passed on the stack. Assume these declarations:

```
struct MyStruct
{
  short a;
  short b;
  short c;
  short d;
  short e;
};

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

The calling function must reserve 12 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register `R1`.

Example 3

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

```
struct MyStruct
{
  int mA;
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value 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 `R1`. The caller assumes that these registers remain untouched. The parameter *x* is passed in `R5`.

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 `R1`, and the return value is returned in `R1`.

FUNCTION DIRECTIVES

Note: This type of directive is primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The IAR C/C++ Compiler for V850 does not use static overlay, because it has no use for it.

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

For more information about the function directives, see the *IAR Assembler Reference Guide for V850*.

Calling functions

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

ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

This section presents the assembler instructions that can be used for calling and returning from functions on the V850 microcontroller.

The normal function calling instruction is the jump-and-link instruction:

```
jarl label, reg
```

The location that the called function should return to (that is, the location immediately after this instruction) is stored in the register.

The destination label must not be further away than 2 Mbytes.

```
jmp [reg]
```

This is an instruction to jump to the location that *reg* points to. After the instruction has been performed, the code located at the label will start executing.

A C function, for instance `alpha`, is represented in assembler as a label with the same name as the function, in this case `alpha`. The location of the label is the actual location of the code of the function.

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

Normal code model

The normal code model requires that the location of the called function must not be further away than 2 Mbytes.

A direct call using this code model is simply:

```
jarl f, lp
```

When a function should return control to the caller, the following instruction will work. This is the same way that functions return regardless of the code model.

```
jmp  [lp]
```

When assigning a function pointer to the location of a function, the following piece of code is used:

```
movhi  hi1(f), R0, R29
movea  lw1(f), R29, R29
```

A function call via function pointers has the following form in this code model:

```
jarl  (?Springboard_R29),lp
```

`?Springboard_R29` is a routine in the runtime library that simply contains the instruction `jmp [R29]`. Because IAR Systems reserves the right to change the runtime environment in the future, it is not recommended to use this function directly in assembler code. Instead, define an equivalent function in C and call it.

Large code model

In this code model the standard jarl instruction cannot be used. Instead, as we can see below, the return address must be computed and explicitly stored in the return register LP.

However, there is one exception to this. If a function call is performed to a static function defined in the same compilation unit (source file), the jarl instruction is used.

The code needed to perform a normal function call is:

```
movhi   hi1(f),zero,R5
movea   lw1(f),R5,R5
jarl    $+4,lp
add     4,lp
jmp     [R5]
```

Line 1 and 2 build the address of the destination f in processor register 5. Line 3 contains the symbol $, which means the current code location. $+4 refers to the location 4 bytes after the jarl instruction, which is where the add instruction is located. Line 3 uses the jarl instruction not as a function call, but as a way to get the current code location to be stored in LP. Line 4 will modify LP so that it will point to the location after the jmp instruction. Finally, line 5 will call the function.

To call a function via function pointers, use the following source code:

```
jarl    $+4,lp
add     4,lp
jmp     [R29]
```

Again, the JARL instruction is used for accessing the current code location in order to compute the return address.

Returning from a function and assigning a function pointer works in the same way as in the normal code model.

Position-independent code model

In the position-independent code model, the generated assembler code must work even if it is placed at a different physical address than the address that the application was built for.

Fortunately, the destination address of the JARL instruction is encoded as the distance from the caller to the destination. This means that we can still use the JARL instruction for plain function calls, because the distance does not change when the code is moved.

Also, functions return the same way as in the normal code model.

For function pointers the situation is not as simple.

In the compiler, the location the function was given by the linker is used as the value of the function pointers. When a function is called via a function pointer, the distance that the code was moved is added to the function pointer, as seen in the following piece of source code:

```
ld.w    (?CODE_DISTANCE-?BREL_BASE-0x8000)[gp],R27
add     R29,R27
jarl    (?Springboard_R27),lp
```

The variable `?CODE_DISTANCE` is like any variable except that it is only accessible in assembler language. Normally, this variable is initialized by taking the difference between the location of a label calculated dynamically at runtime and the location it was assigned by the linker.

See also *Position-independent code*, page 64.

Memory access methods

This section describes the different memory types presented in the chapter *Data storage*. In addition to just 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 V850 instruction set, in particular the different addressing modes used by the instructions that can access memory.

In the V850 architecture, the following instructions are used for accessing memory:

Assembler instruction	Description	Comment
`LD.`*s* `disp16[`*reg*`],` *reg*	Load	
`LD.`*s* `M:disp23[`*reg*`],` *reg*	Load	Only for V850E2M
`ST.`*s* *reg*`, disp16[`*reg*`]`	Store	
`ST.`*s* *reg*`, M:disp23[`*reg*`]`	Store	Only for V850E2M
`SLD.`*s* `disp8[ep],` *reg*	Short load	
`SST.`*s* *reg*`, disp8[ep]`	Short store	
`SET1` *bit#*`, disp16[`*reg*`]`	Sets a bit	
`CLR1` *bit#*`, disp16[`*reg*`]`	Clears a bit	
`NOT1` *bit#*`, disp16[`*reg*`]`	Toggles a bit	
`TST1` *bit#*`, disp16[`*reg*`]`	Tests a bit	

Table 24: Assembler instructions for accessing memory

Explanations of the abbreviations used in the table:

`s`	The size of the operation. This could be `B`, `H`, and `W` for byte, half-word, and word respectively. For the V850E microcontroller, the load instruction could also use `BU` and `HU`. They are used to load a value from the memory while performing a zero extension on the resulting register.
`disp8`	8-, 7-, 5-, or 4-bit unsigned displacement. For `H` and `W` the displacement is 8. For `B` it is 7. In the V850E cores, the `HU` can use 5-bit displacement and `BU` 4 bits.
`disp16`	16-bit signed displacement
`M:disp23`	23-bit signed displacement; only available on V850E2M and above.
`bit#`	A bit number, a constant between 0 and 7
`reg`	Any register
`ep`	Processor register `R30`

A C variable, for instance `alpha`, is accessible in assembly via the assembler label with the same name as the variable, in this case `alpha`. The label is the address of a variable. To access the value of the variable, the memory content location at the address of the label must be accessed, typically loaded into a register.

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;
}
```

NEAR MEMORY ACCESS METHODS

Near memory is the memory that is located at ±32 Kbytes around address 0. The assembler code for storing something using this memory access method is simply:

```
ST.W R1, x[R0]
```

Remember that `R0` will act as a normal processor register with the exception that the value of the register always will be zero.

Clearly, the memory range that this assembler instruction can access is limited by the range of the displacement. Because the displacement is limited to a signed 16-bit value, only the first and last 32 Kbytes of memory can be reached.

Examples

Address of:

```
  MOVEA        MyVar, zero, r1
```

Accessing a global variable:

```
  LD.B         MyVar[zero], r1
```

Accessing a global array using an unknown index:

```
  LD.B         MyArr[r1], r5
```

Accessing a structure using a pointer:

```
  LD.B         (4)[r5],r1
```

BASE-RELATIVE ACCESS METHOD

The base-relative (brel) access method can access two memory areas of 64 Kbytes, one in RAM and one in ROM. Unlike the near memory, the brel memory areas can be placed anywhere in memory.

There are two assembler labels, `?BREL_BASE` and `?BREL_CBASE`, located at the beginning of the brel memory areas. The processor register `R4`—also known as `GP`—and the processor register `R25` are initialized as pointers to a location 32 Kbytes into the memory.

The assembler code for accessing the brel RAM area is:

```
ST.W R5, (x-?BREL_BASE-0x8000)[GP]
```

For the brel RAM area, the displacement used is the value of the expression `(x-?BREL_BASE-0x8000)`, where `x` refers to the absolute location of the variable `x`.

If we add together the value of the global pointer (that points to the location 32 kilobytes after `?BREL_BASE`) with the displacement, we end up at address `x`:

```
= x-?BREL_BASE-0x8000 + GP
= x-?BREL_BASE-0x8000 + ?BREL_BASE+0x8000
= x
```

The assembler code for accessing the brel ROM area is:

```
ST.W R5, (y-?BREL_CBASE-0x8000)[R25]
```

It works the same way as the code for the RAM area.

The base registers `GP` and `R25` do not point to the `?BREL_BASE` label or the `?BREL_CBASE` label, respectively, because the limitation of the displacement of the memory instructions would only let us reach 32 Kbytes into the memory area.

Examples

Address of:

```
MOVEA       (MyVar-?BREL_BASE-0x8000),gp,r1
```

Accessing a global variable:

```
LD.B        (MyVar-?BREL_BASE-0x8000)[gp],r6
```

Accessing a global array using an unknown index:

```
ADD         gp,r1
LD.B        (MyArr-?BREL_BASE-0x8000)[r1],r7
```

Accessing a structure using a pointer:

```
LD.B        (4)[r5],r1
```

BASE-RELATIVE23 ACCESS METHOD

The V850E2M and newer cores provide the 23-bit displacement form of the assembler instructions `LD` and `ST`. The base-relative23 (brel23) access method uses the same base pointers and placeholder segments as the normal base-relative access method, but can access 8 Mbytes of ROM and 8 Mbytes of RAM, respectively.

Examples

Address of:

```
MOVHI       hi1(MyVar),zero,r1
MOVEA       lw1(MyVar),r1,r1
```

Accessing a global variable:

```
LD.BU       M:(MyVar-?BREL_BASE-0x8000)[gp],r6
```

Accessing a global array using an unknown index:

```
ADD         gp,r1
LD.BU       M:(MyArr-?BREL_BASE-0x8000)[r1],r7
```

Accessing a structure using a pointer:

```
LD.B        (4)[r5],r1
```

HUGE ACCESS METHOD

As seen in Table 24, *Assembler instructions for accessing memory*, it is not possible to access an arbitrary location using only one instruction. In the V850 microcontroller, this is solved by using the `MOVHI` instruction to move part of the memory location to access to a temporary register.

For example, to store a variable `x` in huge memory, the following assembler code is used:

```
MOVHI   hi1(x), R0, R5
ST.W    R1, lw1(x)[R5]
```

Examples

Address of:

```
MOVHI       hi1(MyVar),zero,r1
MOVEA       lw1(MyVar),r1,r1
```

Accessing a global variable:

```
MOVHI       hi1(MyVar),zero,r6
LD.B        lw1(MyVar)[r6],r6
```

Accessing a global array using an unknown index:

```
MOVHI       hi1(MyArr),r1,r7
LD.B        lw1(MyArr)[r7],r7
```

Accessing a structure using a pointer:

```
LD.B        (4)[r5],r1
```

SHORT ADDRESSING ACCESS METHOD

The short addressing (saddr) access method uses the short variants of the load and store instructions, `SLD` and `SST`. These instructions are both smaller and faster than the standard `LD` and `ST` instructions.

The assembler label `?SADDR_BASE` is located at the beginning of the 256 byte memory area that can be reached using the short load and store instructions. The processor register `EP` (`R30`) is used as a base pointer to this area. Note that when you are using a data model without short addressing, the `EP` register is instead a pointer to the frame on the stack.

The source code needed to store a value to saddr memory is:

```
SST.W R1, (x-?SADDR_BASE)[ep]
```

As mentioned in Table 24, *Assembler instructions for accessing memory*, this instruction can only reach 256 bytes. When using byte access, it is limited to 128 bytes. This is the reason for the corresponding limitation on the data for the saddr memory type.

NO BIT ACCESS

Data objects declared `__no_bit_access` are never accessed using the V850 bit instructions. This is primarily useful when accessing memory-mapped peripheral units that do not allow bit access.

Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *IAR Assembler Reference Guide for V850*.

CFI DIRECTIVES

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

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

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

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

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

Resource	Description
CFA	The call frames of the stack
R0–R29, EP	Normal registers
LP	The return address register
SP	The stack pointer
EIPC, EIPSW, FEPC, FEPSW, ECR, PSW, CTPC, CTPSW, DBPC, DBPSW, CTBP	Special processor registers
?RET	The return address
BSEL	Bank select register
FPSR	Floating-point status register

Table 25: Call frame information resources defined in a names block

Note: The header file cfi.h contains macros that declare a typical names block and a typical common block. These macros declare a number of resources, both concrete and virtual.

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 test

RSEG CSTACK:DATA:SORT:NOROOT(2)

PUBLIC cfiExample
FUNCTION cfiExample,021203H
ARGFRAME CSTACK, 0, STACK
LOCFRAME CSTACK, 8, STACK

CFI Names cfiNames0
CFI StackFrame CFA SP DATA
CFI Resource R0:32, R1:32, R2:32, SP:32, R4:32, R5:32,
             R6:32, R7:32
CFI Resource R8:32, R9:32, R10:32, R11:32, R12:32,
             R13:32, R14:32
CFI Resource R15:32, R16:32, R17:32, R18:32, R19:32,
             R20:32, R21:32
CFI Resource R22:32, R23:32, R24:32, R25:32, R26:32,
             R27:32, R28:32
CFI Resource R29:32, EP:32, LP:32, EIPC:32, EIPSW:32,
             FEPC:32, FEPSW:32
CFI Resource ECR:32, PSW:32, CTPC:32, CTPSW:32, DBPC:32,
             DBPSW:32
CFI Resource CTBP:32, FPSR:32, BSEL:32
CFI VirtualResource ?RET:32
CFI EndNames cfiNames0
```

```
CFI Common cfiCommon0 Using cfiNames0
CFI CodeAlign 2
CFI DataAlign 4
CFI ReturnAddress ?RET DATA
CFI CFA SP+0
CFI R0 SameValue
CFI R1 Undefined
CFI R2 SameValue
CFI R4 SameValue
CFI R5 Undefined
CFI R6 Undefined
CFI R7 Undefined
CFI R8 Undefined
CFI R9 Undefined
CFI R10 SameValue
CFI R11 SameValue
CFI R12 SameValue
CFI R13 SameValue
CFI R14 SameValue
CFI R15 SameValue
CFI R16 SameValue
CFI R17 SameValue
CFI R18 SameValue
CFI R19 SameValue
CFI R20 SameValue
CFI R21 SameValue
CFI R22 SameValue
CFI R23 SameValue
CFI R24 SameValue
CFI R25 SameValue
CFI R26 SameValue
CFI R27 SameValue
CFI R28 SameValue
CFI R29 SameValue
CFI EP SameValue
CFI LP Undefined
CFI EIPC SameValue
CFI EIPSW SameValue
CFI FEPC SameValue
CFI FEPSW SameValue
CFI ECR SameValue
CFI PSW SameValue
CFI CTPC SameValue
CFI CTPSW SameValue
CFI DBPC SameValue
CFI DBPSW SameValue
CFI CTBP SameValue
```

```
        CFI FPSR SameValue
        CFI BSEL SameValue
        CFI ?RET LP
        CFI EndCommon cfiCommon0

        EXTERN `F`
        FUNCTION `F`,0202H

        RSEG `CODE`:CODE:NOROOT(2)
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample

cfiExample:
        FUNCALL cfiExample, `F`
        LOCFRAME CSTACK, 8, STACK
        PREPARE     {r29,lp},0
        CFI R29 Frame(CFA, -4)
        CFI ?RET Frame(CFA, -8)
        CFI CFA SP+8
        MOV         r1,r29
        JARL        `F`,lp
        ADD         r29,r1
        DISPOSE     0,{r29,lp},[lp]
        CFI EndBlock cfiBlock0

        END
```

Note: The header file `cfi.m85` contains the macros `CFNAMES` and `CFCOMMON`, which declare a typical names block and a typical common block. These two macros declare several resources, both concrete and virtual.

Using C

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

C language overview

The IAR C/C++ Compiler for V850 supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

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

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

- The `inline` keyword advises the compiler that the function defined immediately after the keyword should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a `for` loop
- The `bool` data type
- The `long long` data type
- The `complex` floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator `_Pragma()`
- Variadic macros, which are the preprocessor macro equivalents of `printf` style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option `--vla`
- Inline assembler using the `asm` or the `__asm` keyword, see *Inline assembler*, page 119.

Note: Even though it is a C99 feature, the IAR C/C++ Compiler for V850 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 147. 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 117. 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 291.

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
`--strict`	**Strict**	All IAR C language extensions are disabled; errors are issued for anything that is not part of Standard C.
None	**Standard**	All *extensions to Standard C* are enabled, but no *extensions for embedded systems programming*. For information about extensions, see *IAR C language extensions*, page 147.
`-e`	**Standard with IAR extensions**	All *IAR C language extensions* are enabled.

Table 26: Language extensions

* In the IDE, choose **Project>Options>C/C++ Compiler>Language>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific microcontroller you are using, typically to meet memory restrictions
- *Relaxations to Standard C*—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions, see *Relaxations to Standard C*, page 150.

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 167, and *location*, page 259.

- Alignment control

  Each data type has its own alignment; for more information, see *Alignment*, page 227. If you want to change the alignment, the `#pragma pack` and `#pragma data_alignment` directives are available. If you want to check the alignment of an object, use the `__ALIGNOF__()` operator.

  The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

  - `__ALIGNOF__ (`*`type`*`)`
  - `__ALIGNOF__ (`*`expression`*`)`

  In the second form, the expression is not evaluated.

- Anonymous structs and unions

  C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 165.

- 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 229.

- `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:

`__segment_begin`	Returns the address of the first byte of the named segment.
`__segment_end`	Returns the address of the first byte *after* the named segment.
`__segment_size`	Returns the size of the named segment in bytes.

Note: The aliases `__sfb`, `__sfe`, and `__sfs` can also be used.

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 __near *`.

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

See also *segment*, page 265, and *location*, page 259.

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 233.

- 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 206.

- 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++

IAR Systems supports the C++ language. You can choose between the industry-standard Embedded C++ and Extended Embedded C++. This chapter describes what you need to consider when using the C++ language.

Overview

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, no multiple inheritance, and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

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

Enabling support for C++

In the compiler, the default language is C.

To compile files written in Embedded C++, you must use the `--ec++` compiler option. See *--ec++*, page 206.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See *--eec++*, page 206.

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 V850, 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

```
class MyClass
{
public:
  // Locate a static variable in __memattr memory at address 60
  static __near __no_init int mI @ 0xFFFF8000;


  // A static function using the callt call mechanism
  static __callt void F();

  // A function using the trap call mechanism
  __trap void G();

  // Locate a virtual function in default memory
  virtual 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 95.

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 V850*.

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 154.

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.

Efficient coding for embedded applications

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

The topics discussed are:

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

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

Selecting data types

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

USING EFFICIENT DATA TYPES

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

- 32-bit integers (`int` etc.) are more efficient than 8- and 16-bit integers (`char` and `short`).
- Floating-point types are inefficient. If possible, try to use integers instead. If you have to use floating-point types, notice that 32-bit floating-point numbers are more efficient than 64-bit type doubles. Note that some V850 devices have a floating-point unit which makes floating-point operations faster.

- Use only bitfields with sizes other than 1 bit when you need to optimize the use of data storage. The generated code is both larger and slower than if non-bitfield integers were used.
- Declaring a pointer parameter to `const` data tells the calling function that the data pointed to will not change.

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.

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

By default, a *floating-point constant* in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, 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 *Floating-point types*, page 231.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The V850 microcontroller requires that when accessing data in memory, the data must be aligned. Each element in a structure must be aligned according to its specified type

requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment correct.

There are situations when this can be a problem:

- There are external demands; for example, network communication protocols are usually specified in terms of data types with no padding in between
- You need to save data memory.

For information about alignment requirements, see *Alignment*, page 227.

There are two ways to solve the problem:

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

For more information about the `#pragma pack` directive, see *pack*, page 262.

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 V850 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 206, 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;
  };
} @ 0xFFFF8000;



/* The variables are used here. */
void Test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

This declares an I/O register byte `IOPORT` at address 0. The I/O register has 2 bits declared, `way` and `out`. Note that both the inner structure and the outer union are anonymous.

Anonymous structures and unions are implemented in terms of objects named after the first field, with a prefix `_A_` to place the name in the implementation part of the namespace. In this example, the anonymous union will be implemented through an object named `_A_IOPORT`.

Controlling data and function placement in memory

The compiler provides different mechanisms for controlling placement of functions and data objects in memory. To use memory efficiently, you should be familiar with these mechanisms to know which one is best suited for different situations. You can use:

- Code and data models

  Use the different compiler options for code and data models, respectively, to take advantage of the different addressing modes available for the microcontroller and thereby also place functions and data objects in different parts of memory. For more information about data and code models, see *Data models*, page 44, and *Code models*, page 53, respectively.

- Memory attributes

  Use memory attributes to override the default addressing mode and placement of individual data objects. For more information about memory attributes for data, see *Using data memory attributes*, page 47.

- The `@` operator and the `#pragma location` directive for absolute placement

  Use the `@` operator or the `#pragma location` directive to place individual global and static variables at absolute addresses. The variables must be declared either `__no_init` or `const`.

  This is useful for individual data objects that must be located at a fixed address to conform to external requirements, for example to populate interrupt vectors or other hardware tables. Note that it is not possible to use this notation for absolute placement of individual functions.

- The `@` operator and the `#pragma location` directive for segment placement

  Use the `@` operator or the `#pragma location` directive to place groups of functions or global and static variables in named segments, without having explicit control of each object. The variables must be declared either `__no_init` or `const`. The segments can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the segment begin and end operators. This is also useful if you want an interface between separately linked units, for example an application project and a boot loader project. Use named segments when absolute control over the placement of individual variables is not needed, or not useful.

At compile time, data and functions are placed in different segments, see *Data segments*, page 71, and *Code segments*, page 77, respectively. At link time, one of the most

important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker configuration file, see *Placing segments in memory*, page 68.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

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

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

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

Note: All declarations of variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

Examples

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

```
__no_init volatile char alpha @ 0x1000;/* 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=0x1004
__no_init const int beta;                /* OK */

const int gamma @ 0x1008 = 3;            /* OK */
```

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

These examples show incorrect usage:

```
int delta @ 0x100C;                 /* Error, neither */
                                    /* "__no_init" nor "const".*/

__no_init int epsilon @ 0x100F;     /* 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 `@` 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 place the user-defined segment appropriately in the linker configuration file.

```
__no_init int alpha @ "MY_NOINIT";  /* OK */

#pragma location="MY_CONSTANTS"
const int beta = 42;                /* OK */

const int gamma @ "MY_CONSTANTS" = 17;/* OK */
```

As usual, you can use memory attributes to select a memory for the variable. Note that you must as always place the segment appropriately in the linker configuration file.

```
__huge __no_init int alpha @ "MY_HUGE_NOINIT";/* Placed in
                                                   huge*/
```

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);
```

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 261, for information about the pragma directive.

MULTI-FILE COMPILATION UNITS

In addition to applying different optimizations to different source files or even functions, you can also decide what a compilation unit consists of—one or several source code files.

By default, a compilation unit consists of one source file, but you can also use multi-file compilation to make several source files in a compilation unit. The advantage is that interprocedural optimizations such as inlining, cross call, and cross jump have more source code to work on. Ideally, the whole application should be compiled as one compilation unit. However, for large applications this is not practical because of resource restrictions on the host computer. For more information, see *--mfc*, page 212.

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 205.

OPTIMIZATION LEVELS

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

Optimization level	Description
None (Best debug support)	Variables live through their entire scope Dead code elimination Redundant label elimination Redundant branch elimination
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope
Medium	Same as above, and: Live-dead analysis and optimization Code hoisting Register content analysis and optimization Instruction scheduling (when optimizing for speed) Common subexpression elimination Static clustering
High (Balanced)	Same as above, and: Peephole optimization Cross jumping Cross call (when optimizing for size) Loop unrolling Function inlining Code motion Type-based alias analysis

Table 27: 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 173.

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
- Cross call
- Instruction scheduling.

Common subexpression elimination

Redundant re-evaluation of common subexpressions is by default eliminated at optimization levels Medium and High. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

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

For more information about the command line option, see *--no_cse*, page 214.

Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most efficient for smaller loops, where the loop overhead can be a substantial part of the total loop body.

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

The compiler heuristically decides which loops to unroll. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

For information about the related pragma directive, see *unroll*, page 268. To disable loop unrolling, use the command line option `--no_unroll`, see *--no_unroll*, page 218.

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 63.

Code motion

Evaluation of loop-invariant expressions and common subexpressions are moved to avoid redundant re-evaluation. This optimization, which is performed at optimization level High, normally reduces code size and execution time. The resulting code might however be difficult to debug.

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

For more information about the command line option, see *--no_code_motion*, page 214.

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 217.

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 213.

Cross call

Common code sequences are extracted to local subroutines. This optimization, which is performed at optimization level High, can reduce code size, sometimes dramatically, on behalf of execution time and stack size. The resulting code might however be difficult to debug. This optimization cannot be disabled using the `#pragma optimize` directive.

Note: This option has no effect at optimization levels None, Low, and Medium, unless the option `--do_cross_call` is used.

For more information about related command line options, see *--no_cross_call*, page 214.

Instruction scheduling

The compiler features an instruction scheduler to increase the performance of the generated code. To achieve that goal, the scheduler rearranges the instructions to minimize the number of pipeline stalls emanating from resource conflicts within the microprocessor. Note that not all cores benefit from scheduling. The resulting code might be difficult to debug.

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

For more information about the command line option, see *--no_scheduling*, page 216.

Register locking and register constants

This section introduces the concepts of register locking and register constants.

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

- Some parts of a system could be written in assembler language to improve execution speed. These parts could be given dedicated processor registers.
- The register could be used by an operating system, or by other third-party software.

A register constant is a value that is loaded into a dedicated processor register when the system is initialized. The compiler can then generate code that assumes that the constants are present in the dedicated registers.

REGISTER LOCKING

Register locking is performed using the `--lock_regs` compiler option on predefined groups of registers. See *--lock_regs*, page 211.

Note: The compiler never uses the processor register `R4`.

REGISTER CONSTANTS

When the `--reg_const` option is specified, the compiler will load the value `0xFF` (255 in decimal) and `0xFFFF` (65535 in decimal) into the two processor registers `R18` and `R19`, respectively.

When registers are used as constants, at least two processor registers must be locked with the `--lock_regs` option.

This enables the compiler to generate more efficient code. A typical example is a cast from a 16-bit `unsigned short` to a 32-bit value. (A 16-bit value located in a 32-processor register is assumed to contain garbage in the upper 16 bits.) Using register constants, a two-byte instruction can perform the cast. Without register constants, a

four-byte instruction is required. Note that this is mainly useful for the V850 core, because the V850E and newer cores provide instructions to perform zero extension.

COMPATIBILITY ISSUES

In general, if two modules are used together in the same application, they should have the same setting for register locking and register constants. The reason for this is that registers that can be locked could also be used as parameter registers when calling functions. In other words, the calling convention will depend on the number of locked registers.

However, because this leads to a situation where suppliers of object files and libraries would be forced to make a choice between either delivering many different prebuilt versions or selecting a few configurations to support, there is a compiler option `--lock_regs_compatibility`. Object files compiled using this option can be linked with object files that lock the same or a fewer number of registers. Even files compiled with the `--lock_regs_compatibility` option but not with the `--reg_const` option, can be linked with files that do. See --*lock_regs*, page 211.

To create object files that are compatible with as many options as possible, you should lock 10 registers and specify the `--lock_regs_compatibility` option, using a data model with support for short addressing.

The `--lock_regs_compatibility` compiler option will ensure that files use compatible calling conventions by not allowing functions with too many arguments to be defined or called.

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 174. 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 171.
- Avoid using inline assembler. Instead, try writing the code in C/C++, use intrinsic functions, or write a separate module in assembler language. For more information, see *Mixing C and assembler*, page 117.
- Set the heap size to a value which accommodates the needs of the standard I/O buffer, for example to 1 Kbyte. See *Heap size and standard I/O*, page 76, for more information.

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.

EXTENDING THE CODE SPAN

The Normal and Position-independent code models support a code span of 2 Mbytes. However, in practice it is possible to use more code than this as long as there is no pair of caller and called functions that are further away from each other than 2 Mbytes.

FUNCTION PROTOTYPES

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

- Prototyped

- Kernighan & Ritchie C (K&R C)

Both styles are valid C, however it is strongly recommended to use the prototyped style, and provide a prototype declaration for each public function in a header that is included both in the compilation unit defining the function and in all compilation units using it.

The compiler will not perform type checking on parameters passed to functions declared using K&R style. Using prototype declarations will also result in more efficient code in some cases, as there is no need for type promotion for these functions.

To make the compiler require that all function definitions use the prototyped style, and that all public functions have been declared before being defined, use the Project>Options>C/C++ Compiler>Language 1>Require prototypes compiler option (`--require_prototypes`).

Prototyped style

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

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

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

Kernighan & Ritchie style

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

For example:

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

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

INTEGER TYPES AND BIT NEGATION

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

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 16-bit integer, and two's complement is assumed:

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

Here, the test is always false. On the right hand side, `0x80` is `0x0080`, and `~0x0080` becomes `0xFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 8 bits set.

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code.

For sequences of accesses to variables that you do not want to be interrupted, use the `__monitor` keyword. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. Accessing a small-sized `volatile` variable can be an atomic operation, but you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation. For more information, see *__monitor*, page 246.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 235.

Protecting the eeprom write mechanism

A typical example of when it can be necessary to use the `__monitor` keyword is when protecting the eeprom write mechanism, which can be used from two threads (for example, main code and interrupts). Servicing an interrupt during an EEPROM write sequence can in many cases corrupt the written data.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several V850 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, as in this 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;
} @ 0xFFFF8000;


/* 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 V850 devices. For information about the @ operator, see *Located data*, page 76

NON-INITIALIZED VARIABLES

Normally, the runtime environment will initialize all global and static variables when the application is started.

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in a separate segment, according to the specified memory keyword. See the chapter *Placing code and data* for more information.

For `__no_init`, the `const` keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

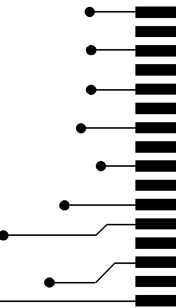
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

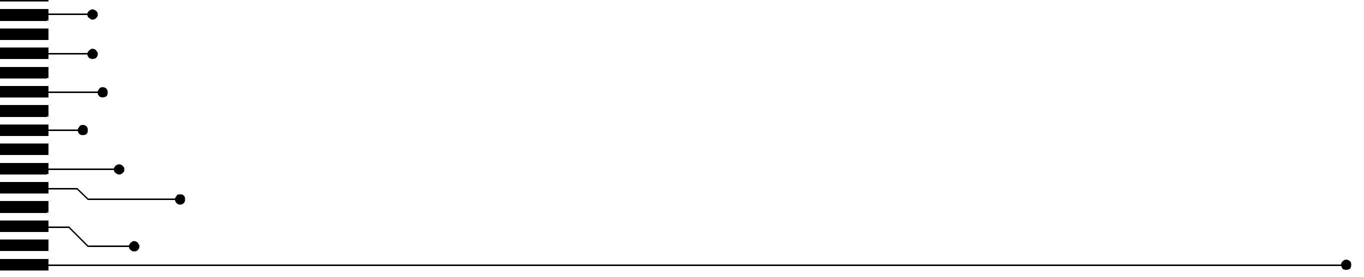
For more information, see *__no_init*, page 247. Note that to use this keyword, language extensions must be enabled; see *-e*, page 206. For more information, see also *object_attribute*, page 261.

Part 2. Reference information

This part of the *IAR C/C++ Compiler Reference Guide for V850* contains these chapters:

- External interface details
- Compiler options
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- Segment reference
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89.





External interface details

This chapter provides reference information about how the compiler interacts with its environment. The chapter briefly lists and describes the invocation syntax, methods for passing options to the tools, environment variables, the include file search procedure, and finally the different types of compiler output.

Invocation syntax

You can use the compiler either from the IDE or from the command line. See the *IDE Project Management and Building Guide* for information about using the compiler from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccv850 [options] [sourcefile] [options]
```

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

```
iccv850 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 `iccv850` command, either before or after the source filename; see *Invocation syntax*, page 185.

- Via environment variables

  The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 186.
- Via a text file, using the `-f` option; see *-f*, page 208.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *Compiler options* chapter.

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
`C_INCLUDE`	Specifies directories to search for include files; for example: `C_INCLUDE=c:\program files\iar systems\embedded workbench 6.`*n*`\v850\inc;c:\headers`
`QCCV850`	Specifies command line options; for example: `QCCV850=-lA asm.lst`

Table 28: 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 209.

  2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 186.

  3 The automatically set up library system include directories. See *--dlib_config*, page 205.
- 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:

```
iccv850 ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

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

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

Note: Both `\` and `/` can be used as directory delimiters.

For information about the syntax for including header files, see *Overview of the preprocessor*, page 281.

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 `r85`.

- Optional list files

  Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 209. 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 189.

- 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 188.

- 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 `--warnings_affect_exit_code` was used.
2	Errors occurred.
3	Fatal errors occurred, making the compiler abort.
4	Internal errors occurred, making the compiler abort.

Table 29: Error return codes

Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

MESSAGE FORMAT

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the compiler is produced in the form:

`filename,linenumber level[tag]: message`

with these elements:

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

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

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code 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 223.

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 page 218.

Error

A diagnostic message that is produced when the compiler finds a construct which clearly violates the C or C++ language rules, such that code cannot be produced. An error will produce a non-zero exit code.

Fatal error

A diagnostic message that is produced when the compiler finds a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the message is issued, compilation terminates. A fatal error will produce a non-zero exit code.

SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all diagnostics messages, except for fatal errors and some of the regular errors.

See *Summary of compiler options*, page 193, 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

This chapter describes the syntax of compiler options and the general syntax rules for specifying option parameters, and gives detailed reference information about each option.

Options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify options from the command line—which is described in more detail in this section—and from within the IDE.

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 185.

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\`:

  ```
  iccv850 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:

  ```
  iccv850 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:

  ```
  iccv850 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:

  ```
  iccv850 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`:

  ```
  iccv850 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
--aggressive_inlining	Inlines functions more aggressively at the optimization level High.
--aggressive_unrolling	Unrolls loops more aggressively at the optimization level High.
--allow_misaligned_data_access	Allows misaligned data accesses for V850E cores

Table 30: Compiler options summary

Command line option	Description
`--c89`	Specifies the C89 dialect
`--char_is_signed`	Treats `char` as signed
`--char_is_unsigned`	Treats `char` as unsigned
`--code_model`	Specifies the code model
`--cpu`	Specifies a specific core
`-D`	Defines preprocessor symbols
`--data_model`	Specifies the data model
`--debug`	Generates debug information
`--dependencies`	Lists file dependencies
`--diag_error`	Treats these as errors
`--diag_remark`	Treats these as remarks
`--diag_suppress`	Suppresses these diagnostics
`--diag_warning`	Treats these as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--disable_sld_suppression`	Disables suppression of `SLD` instructions that could trigger a V850E hardware problem
`--discard_unused_publics`	Discards unused public symbols
`--dlib_config`	Uses the system include files for the DLIB library and determines which configuration of the library to use
`-e`	Enables language extensions
`--ec++`	Specifies Embedded C++
`--eec++`	Specifies Extended Embedded C++
`--enable_multibytes`	Enables support for multibyte characters in source files
`--error_limit`	Specifies the allowed number of errors before compilation stops
`-f`	Extends the command line
`--fpu`	Enables the floating-point unit
`--guard_calls`	Enables guards for function static variable initialization
`--header_context`	Lists all referred source files and header files
`-I`	Specifies include file path

Table 30: Compiler options summary (Continued)

Command line option	Description
-l	Creates a list file
--library_module	Creates a library module
--lock_regs	Locks registers
--lock_regs_compatibility	Permits different register locking levels
-m	Specifies the data model
--macro_positions_in _diagnostics	Obtains positions inside macros in diagnostic messages
--mfc	Enables multi-file compilation
--migration_preprocessor _extensions	Extends the preprocessor
--misrac1998	Enables error messages specific to MISRA-C:1998. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
--misrac2004	Enables error messages specific to MISRA-C:2004. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
--misrac_verbose	*IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
--module_name	Sets the object module name
--no_clustering	Disables static clustering optimizations
--no_code_motion	Disables code motion optimization
--no_cross_call	Disables cross-call optimization
--no_cse	Disables common subexpression elimination
--no_data_model_rt_attribute	Suppresses the generation of the runtime attribute for the data model
--no_inline	Disables function inlining
--no_path_in_file_macros	Removes the path from the return value of the symbols __FILE__ and __BASE_FILE__
--no_scheduling	Disables the instruction scheduler
--no_size_constraints	Relaxes the normal restrictions for code size expansion when optimizing for speed.
--no_static_destruction	Disables destruction of C++ static variables at program exit

Table 30: Compiler options summary (Continued)

Command line option	Description
--no_system_include	Disables the automatic search for system include files
--no_tbaa	Disables type-based alias analysis
--no_typedefs_in_diagnostics	Disables the use of typedef names in diagnostics
--no_unroll	Disables loop unrolling
--no_warnings	Disables all warnings
--no_wrap_diagnostics	Disables wrapping of diagnostic messages
-O	Sets the optimization level
-o	Sets the object filename. Alias for --output.
--omit_types	Excludes type information
--only_stdout	Uses standard output only
--output	Sets the object filename
--predef_macros	Lists the predefined symbols.
--preinclude	Includes an include file before reading the source file
--preprocess	Generates preprocessor output
--public_equ	Defines a global named assembler label
-r	Generates debug information. Alias for --debug.
--reg_const	Puts constants in registers
--relaxed_fp	Relaxes the rules for optimizing floating-point expressions
--remarks	Enables remarks
--require_prototypes	Verifies that functions are declared before they are defined
--silent	Sets silent operation
--strict	Checks for strict compliance with Standard C/C++
--system_include_dir	Specifies the path for system include files
--use_c++_inline	Uses C++ inline semantics in C99
-v	Specifies a specific core
--vla	Enables C99 VLA support
--warnings_affect_exit_code	Warnings affect exit code

Table 30: Compiler options summary (Continued)

Command line option	Description
`--warnings_are_errors`	Warnings are treated as errors

Table 30: 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.

--aggressive_inlining

Syntax `--aggressive_inlining`

Description Use this option to make the compiler inline more functions. This might reduce execution time in some cases, but often increases the code size. The resulting code becomes more difficult to debug.

See also *Inlining functions*, page 63

Project>Options>C/C++ Compier>Optimizations

--aggressive_unrolling

Syntax `--aggressive_unrolling`

Description Use this option to make the compiler perform more aggressive loop unrolling. This might reduce execution time in some cases, but increases the code size. The resulting code becomes more difficult to debug.

See also *Loop unrolling*, page 173

Project>Options>C/C++ Cmpiler>Optimizations

--allow_misaligned_data_access

Syntax `--allow_misaligned_data_access`

Description This option makes it possible to access misaligned data objects on any V850E core. The option can be used in conjunction with the `#pragma pack` directive for `structs` with misaligned members.

Typically, using this option is more efficient than a normal access to a packed structure. However, a misaligned access is slower than an aligned access.

See also *#pragma pack*, page 262 for more information about using the `#pragma pack` directive.

Project>Options>C/C++ Compiler>Optimizations>Enable misaligned data access

--c89

Syntax `--c89`

Description Use this option to enable the C89 C dialect instead of Standard C.

Note: This option is mandatory when the MISRA C checking is enabled.

See also *C language overview*, page 145.

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={normal|pic|large}`

Parameters

`normal` (default)	Allows for up to 2 Mbytes of memory
`pic`	Allows for position-independent code in up to 2 Mbytes of memory
`large`	Allows for up to 4 Gbytes of memory

Description

Use this option to select the code model for which the code is to be generated. The position-independent code (pic) can be placed anywhere in memory. 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, page 53.

Project>Options>General Options>Target>Code model

--cpu

Syntax

`--cpu={v850|v850e|v850e2|v850e2m|v850e2s}`

Parameters

`v850` (default)	Specifies the V850 core
`v850e`	Specifies the V850E and V850ES cores
`v850e2`	Specifies the V850E2 core
`v850e2m`	Specifies the V850E2M core

`v850e2s`	Specifies the V850E2S core

Description

The compiler supports different cores of the V850 microcontroller family. Use this option (or `-v`) to select which of these cores for which the code is to be generated. If you do not choose a processor core, the compiler will compile for the V850 core.

Project>Options>General Options>Target>Device

-D

Syntax

`-D symbol[=value]`

Parameters

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

Description

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

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

```
-Dsymbol
```

is equivalent to:

```
#define symbol 1
```

To get the equivalence of:

```
#define FOO
```

specify the = sign but nothing after, for example:

```
-DFOO=
```

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--data_model, -m

Syntax

`--data_model={t|T|s|S|m|M|l|L}`

Parameters

`t`	Specifies the tiny data model

`T`	Specifies the tiny data model with short addressing
`s` (default)	Specifies the small data model
`S`	Specifies the small data model with short addressing
`m`	Specifies the medium data model
`M`	Specifies the medium data model with short addressing
`l`	Specifies the large data model
`L`	Specifies the large data model with short addressing

Description

Use this option (or `-m`) 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 44.

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

`--dependencies[=[i|m]] {`*`filename`*`|`*`directory`*`}`

Parameters

`i` (default)	Lists only the names of files
`m`	Lists in makefile style

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 192.

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.r85: c:\iar\product\include\stdio.h
foo.r85: 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:

```
%.r85 : %.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++ CompilerLinker>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax `--diagnostics_tables {filename|directory}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 192.

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_sld_suppression

Syntax `--disable_sld_suppression`

Description A hardware conflict can occur in V850E cores for certain code sequences involving the `SLD` instruction. The compiler compensates for this hardware problem automatically by not generating these code sequences. Use this option to override this behavior.

Project>Options>C/C++ Compiler>Code>Disable SLD instruction suppression

--discard_unused_publics

Syntax `--discard_unused_publics`

Description Use this option to discard unused public functions and variables when compiling with the `--mfc` compiler option.

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

See also *--mfc*, page 212 and *Multi-file compilation units*, page 171.

Project>Options>C/C++ Compiler>Discard unused publics

--dlib_config

Syntax `--dlib_config` *`filename`*`.h|`*`config`*

Parameters

`filename`	A DLIB configuration header file. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 192.
`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 `v850\lib`. For examples and information about prebuilt runtime libraries, see *Using a prebuilt library*, page 83.

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 94.

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 147.

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

In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

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

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++

--eec++

Syntax

`--eec++`

Description

In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also

Extended Embedded C++, page 154.

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

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Extended Embedded C++

--enable_multibytes

Syntax

`--enable_multibytes`

Description

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

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

Project>Options>C/C++ Compiler>Language 2>Enable multibyte support

--error_limit

Syntax

`--error_limit=`*n*

Parameters

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

Description

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

This option is not available in the IDE.

-f

Syntax `-f filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 192.

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**.

--fpu

Syntax `--fpu={auto|single|double}`

Parameters

`auto` (default)	Uses the best FPU setting for the selected CPU.
`single`	Uses the floating-point unit for 32-bit operation.
`double`	Uses the floating-point unit for all operations.

Description Use this option to enable the floating-point unit.

Project>Options>General Options>Target>FPU

--guard_calls

Syntax `--guard_calls`

Description Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.

Note: This option requires a threaded C++ environment, which is not supported in the IAR C/C++ Compiler for V850.

This option is not available in the IDE.

--header_context

Syntax `--header_context`

Description

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

This option is not available in the IDE.

-I

Syntax `-I` *`path`*

Parameters

`path`	The search path for `#include` files

Description

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

See also

Include file search procedure, page 186.

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 `-la`, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *
`B`	Basic assembler list file. This file has the same contents as a list file produced with `-lA`, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
`c`	C or C++ list file
`C` (default)	C or C++ list file with assembler source as comments
`D`	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
`N`	No diagnostics in file
`H`	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

* This makes the list file less useful as input to the assembler, but more useful for reading by a human.

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 192.

Description

Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.

To set related options, choose:

Project>Options>C/C++ Compiler>List

--library_module

Syntax

`--library_module`

Description

Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.

Project>Options>C/C++ Compiler>Output>Module type>Library Module

--lock_regs

Syntax `--lock_regs={0|2|6|10}`

Parameters

`0`	No registers are locked
`2`	Locks registers `R18-R19`
`6`	Locks registers `R17-R22`
`10`	Locks registers `R15-R24`

Description Normally `R1` and `R5–R29` are available for the compiler to use.

Use this option to prevent the compiler from using a specific set of registers.

Note that the register `R2` is free to use by an operating system, because it is not used by the compiler at all.

See also *Register locking and register constants*, page 176 and *--lock_regs_compatibility*, page 211.

Project>Options>C/C++ Compiler>Code>Use of registers

--lock_regs_compatibility

Syntax `--lock_regs_compatibility`

Description Use this option to be able to link the module being compiled with object files that lock fewer registers than the module. Those object files can also use register constants, even if no register constants are used for this module.

This option does not allow definitions or functions that are not compatible when different register locking levels are used. In practice this means an upper limit to the number of parameters to functions.

The `--lock_regs_compatibility` option is well suited for use by a third-party library provider to keep down the number of required configurations.

See also *Compatibility issues*, page 177 and *--lock_regs*, page 211.

Project>Options>C/C++ Compiler>Code>Use of register

--macro_positions_in_diagnostics

Syntax

```
--macro_positions_in_diagnostics
```

Description

Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.

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

--mfc

Syntax

```
--mfc
```

Description

Use this option to enable *multi-file compilation*. This means that the compiler compiles one or several source files specified on the command line as one unit, which enhances interprocedural optimizations.

Note: The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the `-o` compiler option and specify a certain output file.

Example

```
iccv850 myfile1.c myfile2.c myfile3.c --mfc
```

See also

--*discard_unused_publics*, page 205, --*output, -o*, page 220, and *Multi-file compilation units*, page 171.

Project>Options>C/C++ Compiler>Multi-file compilation

--migration_preprocessor_extensions

Syntax

```
--migration_preprocessor_extensions
```

Description

If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you might want to use this option. Use this option to use the following in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

Note: If you use this option, not only will the compiler accept code that does not conform to the ISO/ANSI C standard, but it will also reject some code that does conform to the standard.

Important! Do not depend on these extensions in newly written code, because support for them might be removed in future compiler versions.

Project>Options>C/C++ Compiler>Language>Enable IAR migration preprocessor extensions

--module_name

Syntax `--module_name=*name*`

Parameters

name	An explicit object module name

Description

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.

This option is useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

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

--no_clustering

Syntax `--no_clustering`

Description

Use this option to disable static clustering optimizations.

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

See also *Static clustering*, page 175.

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 174.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cross_call

Syntax `--no_cross_call`

Description Use this option to disable the cross-call optimization.

Note: This option has no effect at optimization levels below High, or when optimizing Balanced or for Speed, because cross-call optimization is not enabled then.

See also *Cross call*, page 175 .

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

--no_cse

Syntax `--no_cse`

Description Use this option to disable common subexpression elimination.

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

See also *Common subexpression elimination*, page 173.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_data_model_rt_attribute

Syntax `--no_data_model_rt_attribute`

Description Suppresses the generation of the runtime attribute for the data model. This is useful when compiling a file that might be used together with other files built using other data models, for example when providing a third-party library.

Note that great care must be taken to ensure that nothing that is provided to other modules rely on the selected data model, in particular global variables without an explicit memory attribute.

For example, the prebuilt libraries are built using this option.

To set this option, use **Project>Options>C/C++ Compiler>Extra options**

--no_inline

Syntax `--no_inline`

Description Use this option to disable function inlining.

See also *Inlining functions*, page 63.

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 282.

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 176.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling

--no_size_constraints

Syntax `--no_size_constraints`

Description Use this option to relax the normal restrictions for code size expansion when optimizing for high speed.

Note: This option has no effect unless used with `-Ohs`.

See also *Speed versus size*, page 173.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>No size constraints

--no_static_destruction

Syntax `--no_static_destruction`

Description Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.

Use this option to suppress the emission of such code.

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

--no_system_include

Syntax `--no_system_include`

Description By default, the compiler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the `-I` compiler option.

See also --*dlib_config*, page 205, and --*system_include_dir*, page 224.

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 174.

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

--no_typedefs_in_diagnostics

Syntax `--no_typedefs_in_diagnostics`

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

Example

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

will give an error message like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

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

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

--no_unroll

Syntax

`--no_unroll`

Description

Use this option to disable loop unrolling.

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

See also

Loop unrolling, page 173.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax

`--no_warnings`

Description

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

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax

`--no_wrap_diagnostics`

Description

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

This option is not available in the IDE.

-O

Syntax `-O[n|l|m|h|hs|hz]`

Parameters

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

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

Description

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

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

See also

Controlling compiler optimizations, page 170.

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

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 192.

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 `r85`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--predef_macros

Syntax

`--predef_macros {`*`filename`*`|`*`directory`*`}`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 192.

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 `--preinclude includefile`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 192.

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 `--preprocess[=[c][n][l]] {filename|directory}`

Parameters

`c`	Preserve comments
`n`	Preprocess only
`l`	Generate `#line` directives

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 192.

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 `--public_equ symbol[=value]`

Parameters

symbol	The name of the assembler symbol to be defined
value	An optional value of the defined assembler symbol

Description

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

This option is not available in the IDE.

--reg_const

Syntax

```
--reg_const
```

Description

Use this option with the V850 core (--cpu=v850) to permanently load the two numbers 255 (0xFF) and 65535 (0xFFFF) into the two dedicated registers R18 and R19. This enables the compiler to generate more efficient code.

Note: This option should not be used with the V850E core and higher.

Example

On the V850 microcontroller, the 4-byte instruction is normally used for zero-extending a character to full register width:

```
ANDI     0x000000FF,R1,R1
```

When the --reg_const option is used, the following 2-byte instruction is generated instead:

```
AND      R18, R1
```

To use this feature, at least two registers must be locked with the --lock_regs option.

See also

Register locking and register constants, page 176.

Project>Options>C/C++ Compiler>Code>Use of registers

--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 189.

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 147.

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. For information about specifying a path, see *Rules for specifying a filename or directory as parameters*, page 192.

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 205, and *--no_system_include*, page 217.

This option is not available in the IDE.

--use_c++_inline

Syntax

```
--use_c++_inline
```

Description

Standard C uses slightly different semantics for the `inline` keyword than C++ does. Use this option if you want C++ semantics when you are using C.

See also

Inlining functions, page 63

Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

-v

Syntax

```
-v={0|1|2|3|4}
```

Parameters

`0` (default)	Specifies the V850 core
`1`	Specifies the V850E and V850ES cores
`2`	Specifies the V850E2 core
`3`	Specifies the V850E2M core
`4`	Specifies the V850E2S core

Description

The compiler supports different cores of the V850 microcontroller family. Use this option (or `--cpu`) to select which of these cores for which the code is to be generated. If you do not choose a processor core, the compiler will compile for the V850 core.

Project>Options>General Options>Target>Device

--vla

Syntax `--vla`

Description Use this option to enable support for C99 variable length arrays. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the `--c89` compiler option.

Note: `--vla` should not be used together with the `longjmp` library function, as that can lead to memory leakages.

See also *C language overview*, page 145.

Project>Options>C/C++ Compiler>Language 1>C dialect>Allow VLA

--warnings_affect_exit_code

Syntax `--warnings_affect_exit_code`

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

This option is not available in the IDE.

--warnings_are_errors

Syntax `--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 204.

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

Data representation

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

See the chapter *Efficient coding for embedded applications* for information about which data types 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`. However, the code needed to access a packed structure is typically much more inefficient than the code for accessing an aligned structure.

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 234.

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

ALIGNMENT ON THE V850 MICROCONTROLLER

The V850 microcontroller can access 4-byte objects using a single assembler instruction only when the object is stored at an address divisible by 4. For the same reason, 2-byte objects must be stored at addresses divisible by 2.

Basic data types

The compiler supports both all Standard C basic data types and some additional types.

INTEGER TYPES

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

Data type	Size	Range	Alignment
bool	8 bits	0 to 1	1
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
signed short	16 bits	-32768 to 32767	2
unsigned short	16 bits	0 to 65535	2
signed int	32 bits	-32768 to 32767	4
unsigned int	32 bits	0 to 65535	4
signed long	32 bits	-2^{31} to 2^{31}-1	4
unsigned long	32 bits	0 to 2^{32}-1	4
signed long long	64 bits	-2^{63} to 2^{63}-1	4
unsigned long long	64 bits	0 to 2^{64}-1	4

Table 31: Integer types

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

Bool

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

The long long type

The `long long` data type is supported with this restriction:

A `long long` variable cannot be used in a switch statement.

The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

When IAR Systems language extensions are enabled, and in C++, the enum constants and types can also be of the type long, unsigned long, long long, or unsigned long long .

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

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

The char type

The char type is by default unsigned in the compiler, but the --char_is_signed compiler option allows you to make it signed. Note, however, that the library is compiled with the char type as unsigned.

The wchar_t type

The wchar_t data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The wchar_t data type is supported by default in the C++ language. To use the wchar_t type also in C source code, you must include the file stddef.h from the runtime library.

Bitfields

In 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 V850, plain integer types are treated as signed.

Bitfields in expressions are treated as int if int can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.

Each bitfield is placed into a container of its base type from the least significant bit to the most significant bit. If the last container is of the same type and has enough bits available, the bitfield is placed into this container, otherwise a new container is allocated.

If you use the directive #pragma bitfield=reversed, bitfields are placed from the most significant bit to the least significant bit in each container. See *bitfields*, page 253.

Example

Assume this example:

```
struct BitfieldExample
{
  uint32_t a : 12;
  uint16_t b : 3;
  uint16_t c : 7;
  uint8_t  d;
};
```

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset `0` and puts `a` into the least significant 12 bits of the container.

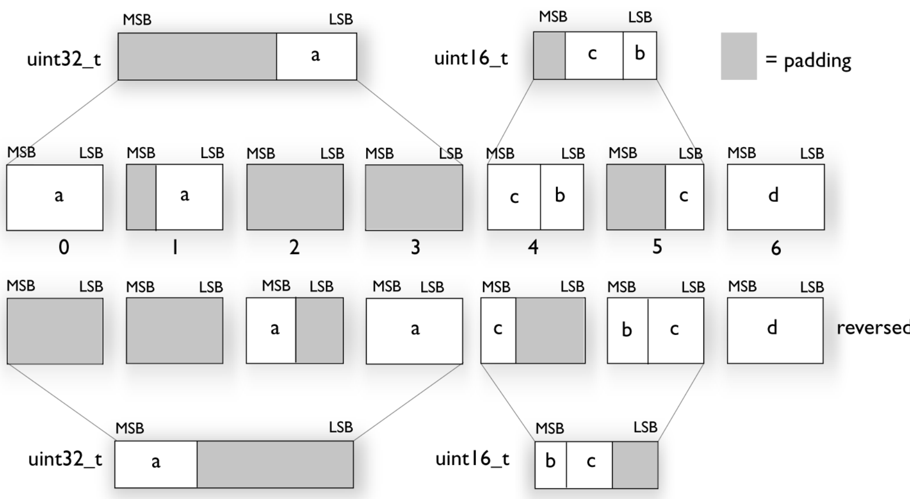
To place the second bitfield, `b`, a new container is allocated at offset `4`, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

The fourth member, `d`, is allocated into the byte at offset `6`. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order , each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example` :



FLOATING-POINT TYPES

In the IAR C/C++ Compiler for V850, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

Type	Size	Alignment
`float`	32 bits	4
`double`	64 bits	4
`long double`	64 bits	4

Table 32: Floating-point types

The compiler supports subnormal numbers, but not the floating-point unit (FPU). When using the FPU, operations that should produce subnormal numbers will instead generate zero.

Floating-point environment

Exception flags are not supported. The `feraiseexcept` function does not raise any exceptions.

32-bit floating-point format

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

31	30 23	22 0
S	Exponent	Mantissa

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

The value of the number is:

$(-1)^{S} * 2^{(Exponent-127)} * 1.Mantissa$

The range of the number is at least:

`±1.18E-38 to ±3.39E+38`

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 7 decimal digits.

64-bit floating-point format

The representation of a 64-bit floating-point number as an integer is:

63	62 52	51 0
S	Exponent	Mantissa

The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

$(-1)^S * 2^{(Exponent-1023)} * 1.Mantissa$

The range of the number is at least:

±2.23E-308 to ±1.79E+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 and 1023 for 32-bit and 64-bit floating-point values, respectively.

Note: The floating-point unit (FPU) does not support subnormal numbers. Instead, operations that should have resulted in a subnormal number return zero.

Pointer types

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

FUNCTION POINTERS

The size of function pointers is always 32 bits, and they can address the entire memory.

DATA POINTERS

The size of data pointers is always 32 bits, and they can address the entire memory.

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a value of an integer type to a pointer of a larger type is performed by 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 required to hold the maximum size of an object. In the IAR C/C++ Compiler for V850, the size of `size_t` is 32 bits.

ptrdiff_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the IAR C/C++ Compiler for V850, the size of `ptrdiff_t` is 32 bits.

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for V850, the size of `intptr_t` is 32 bits.

uintptr_t

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

Structure types

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

ALIGNMENT

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

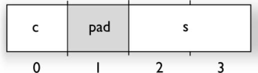
GENERAL LAYOUT

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

Example

```
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.

Example

This example declares a packed structure:

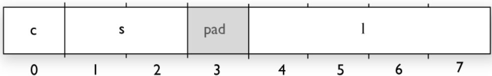
```
#pragma pack(1)
struct S
{
  char c;
  short s;
};

#pragma pack()
```

In this example, the structure S has this memory layout:

This 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;
};
```

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 164.

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 V850 are described below.

Rules for accesses

In the IAR C/C++ Compiler for V850, accesses to `volatile` declared objects are subject to these rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The compiler adheres to these rules for all accesses to 8-bit, 16-bit, and 32-bit objects.

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` and located in all memory types except saddr are allocated in read-only memory. For saddr, the objects are allocated in RAM and initialized by the runtime system at startup.

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

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions

can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.

Extended keywords

This chapter describes the extended keywords that support specific features of the V850 microcontroller and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

For information about the address ranges of the different memory areas, see the chapter *Segment reference*.

General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The compiler provides a set of attributes that can be used on functions or data objects to support specific features of the V850 microcontroller. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For more information about each attribute, see *Descriptions of extended keywords*, page 243.

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 206 for more information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes explicitly in your declarations, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

- Available *data memory attributes*: `__near`, `__brel`, `__brel23`, `__huge`, and `__saddr`.

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

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt`, `__monitor`, `__task`, `__trap`, `__callt`, and `__syscall`
- *Data type attributes*: `__no_bit_access`.

You can specify as many type attributes as required for each level of pointer indirection.

Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the`__huge` type attribute to the variables `i` and `j`; in other words, the variables `i` and `j` are placed in huge memory. However, note that an individual member of a `struct` or `union` cannot have a type attribute. The variables `k` and `l` behave in the same way:

```
__huge int i, j;
int __huge k, l;
```

Note that the attribute affects both identifiers.

This declaration of i and j is equivalent with the previous one:

```
#pragma type_attribute=__huge
int i, j;
```

The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Note that the pragma directive has no effect if a memory attribute is already explicitly declared.

For more examples of using memory attributes, see *More examples*, page 49.

An easier way of specifying storage is to use type definitions. These two declarations are equivalent:

```
typedef char __huge Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__huge char b;
char __huge *bp;
```

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

Syntax for type attributes on functions

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

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

or

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

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

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

OBJECT ATTRIBUTES

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__flat`, `__intrinsic`, `__noreturn`, and `vector`.

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 167. For more information about `vector`, see *vector*, page 269.

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
`__brel`	Controls the storage of data objects
`__brel23`	Controls the storage of data objects
`__callt`	Controls the storage of functions for the V850E cores
`__flat`	Inhibits saving certain processor registers
`__huge`	Controls the storage of data objects
`__interrupt`	Specifies interrupt functions
`__intrinsic`	Reserved for compiler internal use only
`__monitor`	Specifies atomic execution of a function

Table 33: Extended keywords summary

Extended keyword	Description
`__near`	Controls the storage of data objects
`__no_bit_access`	Prevents bit-instruction accessing of data objects
`__no_init`	Places a data object in non-volatile memory
`__noreturn`	Informs the compiler that the function will not return
`__root`	Ensures that a function or variable is included in the object code even if unused
`__saddr`	Controls the storage of data objects
`__syscall`	Controls the storage of functions for the V850E cores
`__task`	Relaxes the rules for preserving registers
`__trap`	Supports trap functions

Table 33: Extended keywords summary (Continued)

Descriptions of extended keywords

These sections give detailed information about each extended keyword.

__brel

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description

The `__brel` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in base-relative (brel) memory.

Storage information

Address range: 64 Kbytes anywhere in RAM and 64 Kbytes anywhere in ROM

Example

```
__brel int x;
```

See also

Memory types, page 45.

__brel23

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description — The `__brel23` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in base-relative (brel) memory using the disp23 addressing mode of `LD` and `ST` instructions.

This memory attribute is only available for the V850E2M core and above.

Storage information — Address range: 8 Mbytes anywhere in RAM and 8 Mbytes anywhere in ROM. The memory is overlayed with the normal brel memory.

Example

```
__brel23 int x;
```

See also — *Memory types*, page 45 and *Memory access methods*, page 134.

__callt

Syntax — Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description — All V850E cores have a *call table* that can be used for storing 64 function pointers. Calls to such functions can then be performed by executing a `CALLT` instruction.

The `__callt` memory attribute places individual functions in the call table.

A call table function which is declared without a `#pragma vector` directive cannot be called directly from a C/C++ program.

Storage information — Vector range: 0-63

Example — Declaring a callt function using call table vector `0x25`:

```
#pragma vector=0x25
__callt void my_callt_function(int my_int);
```

See also — *Callt functions*, page 58.

__flat

Syntax — Follows the generic syntax rules for object attributes, see *Object attributes*, page 241.

Description — The `__flat` keyword can be used in the definition of a trap, interrupt, callt, or syscall function to inhibit the generation of code that stores and restores special processor registers at the entry and leave code, respectively.

Note: You should not use this keyword if the system is supposed to handle nested interrupts, or if the trap or callt function could—directly or indirectly—call another trap or callt function.

Example

```
__flat __interrupt void my_int_func(void)
```

See also

Primitives for interrupts, concurrency, and OS-related programming, page 55

__huge

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description

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

Storage information

- Address range: Anywhere in memory
- Maximum object size: 4 Gbytes

Example

```
__huge int x;
```

See also

Memory types, page 45.

__interrupt

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description

The `__interrupt` keyword specifies interrupt functions. To specify one or several interrupt vectors, use the `#pragma vector` directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a `void` return type and cannot have any parameters.

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

Example

```
#pragma vector=0x70
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 55, *vector*, page 269, and *INTVEC*, page 310.

__intrinsic

Description

The `__intrinsic` keyword is reserved for compiler internal use only.

__monitor

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description

The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.

Example

```
__monitor int get_lock(void);
```

See also

Monitor functions, page 59. For information about related intrinsic functions, see *__disable_interrupt*, page 274, *__enable_interrupt*, page 274, *__get_interrupt_state*, page 275, and *__set_interrupt_state*, page 279, respectively.

__near

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description

The `__near` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in near memory. You can also use the `__near` attribute to create a pointer explicitly pointing to an object located in the near memory.

Storage information

Address range: `0x0-07FFF` and `0xFFFF8000-0xFFFFFFFF` (64 Kbytes)

Example

```
__near int x;
```

See also

Memory types, page 45.

__no_bit_access

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description

Data objects declared with the `__no_bit_access` keyword will not be accessed using bit instructions. The main use of this keyword is to declare memory-mapped peripheral units that do not support bit access.

Example

```
__no_bit_access int x;
```

__no_init

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 241.

Description

Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Example

```
__no_init int myarray[10];
```

__noreturn

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 241.

Description

The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

Example

```
__noreturn void terminate(void);
```

__root

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 241.

Description

A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.

Example

```
__root int myarray[10];
```

See also

For more information about modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide.*

__saddr

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 239.

Description

The `__saddr` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in short addressing (saddr) memory. You can also use the `__saddr` attribute to create a pointer explicitly pointing to an object located in the short addressing memory.

Storage information

- Address range: From processor register `R30` (`EP`) and 256 bytes onward.
- Maximum object size: 256 bytes (128 bytes for objects that require byte access).

Example

```
__saddr int x;
```

See also

Memory types, page 45.

__syscall

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description

All V850E2M and newer cores have a *syscall table* that can be used for storing 256 function pointers. Calls to such functions can then be performed by executing a `SYSCALL` instruction.

The `__syscall` memory attribute places individual functions in the syscall table.

A syscall table function which is declared without a `#pragma vector` directive cannot be called directly from a C/C++ program.

Storage information

Vector range: 0-255

Example

Declaring a syscall function using syscall table vector `0x25`:

```
#pragma vector=0x25
__syscall void my_syscall_function(int my_int);
```

See also

Syscall functions, page 58.

__task

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description

This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the start function for a task in an RTOS.

By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared `__task` do not save all registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared `__task`.

Example

```
__task void my_handler(void);
```

__trap

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 239.

Description

A trap function is called and then executed by the `TRAP` assembler instruction and returned by the `EIRET` or `RETI` instruction, depending on which core you are using. To specify one or several vectors, use the `#pragma vector` directive. See the chip manufacturer's hardware documentation for information about the trap vector range. If a trap vector is not given, an error will be issued if the function is called. A trap function can take parameters and return a value and it has the same calling convention as other functions. You can call the trap functions from your C or C++ application.

Example

```
#pragma vector=0x0B
__trap int my_trap_function(void);
```

See also

Calling convention, page 124 and *Trap functions*, page 57.

Pragma directives

This chapter describes the pragma directives of the compiler.

The #pragma directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
`bitfields`	Controls the order of bitfield members
`constseg`	Places constant variables in a named segment.
`data_alignment`	Gives a variable a higher (more strict) alignment
`dataseg`	Places variables in a named segment.
`diag_default`	Changes the severity level of diagnostic messages
`diag_error`	Changes the severity level of diagnostic messages
`diag_remark`	Changes the severity level of diagnostic messages
`diag_suppress`	Suppresses diagnostic messages
`diag_warning`	Changes the severity level of diagnostic messages
`error`	Signals an error while parsing
`include_alias`	Specifies an alias for an include file
`inline`	Controls inlining of a function
`language`	Controls the IAR Systems language extensions
`location`	Specifies the absolute address of a variable, or places groups of functions or variables in named segments

Table 34: Pragma directives summary

Pragma directive	Description
message	Prints a message
no_epilogue	Inlines the prologue and epilogue sequences
object_attribute	Changes the definition of a variable or a function
optimize	Specifies the type and level of an optimization
pack	Specifies the alignment of structures and union members
__printf_args	Verifies that a function with a printf-style format string is called with the correct arguments
required	Ensures that a symbol that is needed by another symbol is included in the linked output
rtmodel	Adds a runtime model attribute to the module
__scanf_args	Verifies that a function with a scanf-style format string is called with the correct arguments
section	This directive is an alias for #pragma segment
segment	Declares a segment name to be used by intrinsic functions
STDC CX_LIMITED_RANGE	Specifies whether the compiler can use normal complex mathematical formulas or not
STDC FENV_ACCESS	Specifies whether your source code accesses the floating-point environment or not.
STDC FP_CONTRACT	Specifies whether the compiler is allowed to contract floating-point expressions or not.
type_attribute	Changes the declaration and definitions of a variable or function
unroll	Unrolls loops
vector	Specifies the vector of an interrupt or trap function

Table 34: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 324.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

bitfields

Syntax

```
#pragma bitfields={reversed|default}
```

Parameters

`reversed`	Bitfield members are placed from the most significant bit to the least significant bit.
`default`	Bitfield members are placed from the least significant bit to the most significant bit.

Description

Use this pragma directive to control the order of bitfield members.

Example

```
#pragma bitfields=reversed
/* Structure that uses reversed bitfields. */
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 229.

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=__near MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

data_alignment

Syntax

`#pragma data_alignment=`*`expression`*

Parameters

expression	A constant which must be a power of two (1, 2, 4, etc.).

Description

Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

Note: Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

dataseg

Syntax

`#pragma dataseg=[`*`__memoryattribute`*`]{`*`SEGMENT_NAME`*`|default}`

Parameters

__memoryattribute	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
SEGMENT_NAME	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment.

Description

Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared `__no_init`. The setting remains active until you turn it off again with the `#pragma dataseg=default` directive.

Example

```
#pragma dataseg=__near MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

diag_default

Syntax

`#pragma diag_default=`*`tag`*`[,`*`tag`*`,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `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 189.

diag_error

Syntax

`#pragma diag_error=`*`tag`*`[,`*`tag`*`,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`.

Description

Use this pragma directive to change the severity level to `error` for the specified diagnostics.

See also

Diagnostics, page 189.

diag_remark

Syntax
```
#pragma diag_remark=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`.

Description
Use this pragma directive to change the severity level to `remark` for the specified diagnostic messages.

See also
Diagnostics, page 189.

diag_suppress

Syntax
```
#pragma diag_suppress=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description
Use this pragma directive to suppress the specified diagnostic messages.

See also
Diagnostics, page 189.

diag_warning

Syntax
```
#pragma diag_warning=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`.

Description
Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages.

See also
Diagnostics, page 189.

error

Syntax

```
#pragma error message
```

Parameters

message — A string that represents the error message.

Description

Use this pragma directive to cause an error message when it is parsed. This mechanism is different from the preprocessor directive `#error`, because the `#pragma error` directive can be included in a preprocessor macro using the `_Pragma` form of the directive and only causes an error if the macro is used.

Example

```
#if FOO_AVAILABLE
#define FOO ...
#else
#define FOO _Pragma("error\"Foo is not available\"")
#endif
```

If `FOO_AVAILABLE` is zero, an error will be signaled if the `FOO` macro is used in actual source code.

include_alias

Syntax

```
#pragma include_alias ("orig_header" , "subst_header")
#pragma include_alias (<orig_header> , <subst_header>)
```

Parameters

orig_header — The name of a header file for which you want to create an alias.

subst_header — The alias for the original header file.

Description

Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

See also

Include file search procedure, page 186.

inline

Syntax

```
#pragma inline[=forced|=never]
```

Parameters

No parameter	Has the same effect as the `inline` keyword.
`forced`	Disables the compiler's heuristics and forces inlining.
`never`	Disables the compiler's heuristics and makes sure that the function will not be inlined.

Description

Use `#pragma inline` to advise the compiler that the function defined immediately after the directive should be inlined according to C++ inline semantics.

Specifying `#pragma inline=forced` will always inline the defined function. If the compiler fails to inline the function for some reason, for example due to recursion, a warning message is emitted.

Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` will enable inlining of the function also on the Medium optimization level.

See also

Inlining functions, page 63

language

Syntax

```
#pragma language={extended|default|save|restore}
```

Parameters

`extended`	Enables the IAR Systems language extensions from the first use of the pragma directive and onward.
`default`	From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.

`save|restore`	Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code. Each use of `save` must be followed by a matching `restore` in the same file without any intervening `#include` directive.

Description

Use this pragma directive to control the use of language extensions.

Example

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 206 and --*strict*, page 224.

location

Syntax

`#pragma location={`*`address`*`|`*`NAME`*`}`

Parameters

address	The absolute address of the global or static variable for which you want an absolute location.
NAME	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

Description

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive. 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=0xFF2000
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0xFF2000 */

#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 167.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to the standard output stream.

Description

Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

no_epilogue

Syntax

```
#pragma no_epilogue
```

Description

Use this pragma directive to inline the prologue and epilogue sequences instead of performing calls to internal library routines. This pragma directive can be used when a function needs to exist on its own as in, for example, a bootloader that needs to be independent of the libraries it is replacing.

Example

```
#pragma no_epilogue
void bootloader(void) @"BOOTSECTOR"
{
    ...
}
```

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[,object_attribute,...]
```

Parameters

For information about object attributes that can be used with this pragma directive, see *Object attributes*, page 241.

Description

Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.

Example

```
#pragma object_attribute=__no_init
char bar;
```

See also

General syntax rules for extended keywords, page 239.

optimize

Syntax

```
#pragma optimize=[goal][level][no_optimization...]
```

Parameters

goal — Choose between:

`size`, optimizes for size

`balanced`, optimizes balanced between speed and size

`speed`, optimizes for speed.

`no_size_constraints`, optimizes for speed, but relaxes the normal restrictions for code size expansion.

level — Specifies the level of optimization; choose between `none`, `low`, `medium`, or `high`.

`no_optimization`	Disables one or several optimizations; choose between: `no_code_motion`, disables code motion `no_cse`, disables common subexpression elimination `no_inline`, disables function inlining `no_tbaa`, disables type-based alias analysis `no_unroll`, disables loop unrolling `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 173.

pack

Syntax

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop}[,name] [,n])
```

Parameters

n	Sets an optional structure alignment; one of: `1`, `2`, `4`, `8`, or `16`
Empty list	Restores the structure alignment to default
`push`	Sets a temporary structure alignment
`pop`	Restores the structure alignment from a temporarily pushed alignment
name	An optional pushed or popped alignment label

Description

Use this pragma directive to specify the maximum alignment of `struct` and `union` members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or the end of the compilation unit.

Note: This can result in significantly larger and slower code when accessing members of the structure.

See also

Structure types, page 233.

__printf_args

Syntax

```
#pragma __printf_args
```

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __printf_args
int printf(char const *,...);

void PrintNumbers(unsigned short x)
{
  printf("%d", x);  /* Compiler checks that x is an integer */
}
```

required

Syntax

```
#pragma required=symbol
```

Parameters

symbol	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.

Example

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
  /* Do something here. */
}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax

```
#pragma rtmodel="key","value"
```

Parameters

"*key*"	A text string that specifies the runtime model attribute.
"*value*"	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value `*`. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 112.

__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

NAME	The name of the segment.

`__memoryattribute`	An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.
`align`	Specifies an alignment for the segment. The value must be a constant integer expression to the power of two.

Description

Use this pragma directive to define a segment name that can be used by the segment operators `__segment_begin`, `__segment_end`, and `__segment_size`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

The *`align`* and the *`__memoryattribute`* parameters are only relevant when used together with the segment operators `__segment_begin`, `__segment_end`, and `__segment_size`. If you consider using *`align`* on an individual variable to achieve a higher alignment, you must instead use the `#pragma data_alignment` directive.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

`void __memoryattribute *.`

Example

```
#pragma segment="MYSEG" __huge 4

/* Fill MYSEG with zeroes */
void clear(void)
{
  for (long * p = __sfb("MYSEG"); p != __sfe("MYSEG"); ++p)
    *p = 0;
}
```

See also

Dedicated segment operators, page 149. For more information about segments, see the chapter *Placing code and data*.

STDC CX_LIMITED_RANGE

Syntax

```
#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}
```

Parameters

`ON`	Normal complex mathematic formulas can be used.
`OFF`	Normal complex mathematic formulas cannot be used.
`DEFAULT`	Sets the default behavior, that is `OFF`.

Description

Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for `*` (multiplication), `/` (division), and `abs`.

Note: This directive is required by Standard C. The directive is recognized but has no effect in the compiler.

STDC FENV_ACCESS

Syntax

```
#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}
```

Parameters

`ON`	Source code accesses the floating-point environment. Note that this argument is not supported by the compiler.
`OFF`	Source code does not access the floating-point environment.
`DEFAULT`	Sets the default behavior, that is `OFF`.

Description

Use this pragma directive to specify whether your source code accesses the floating-point environment or not.

Note: This directive is required by Standard C.

STDC FP_CONTRACT

Syntax

```
#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}
```

Parameters

`ON`	The compiler is allowed to contract floating-point expressions.
`OFF`	The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler.
`DEFAULT`	Sets the default behavior, that is `ON`.

Description

Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C.

Example

```
#pragma STDC FP_CONTRACT=ON
```

type_attribute

Syntax

```
#pragma type_attribute=type_attribute[,type_attribute,...]
```

Parameters

For information about type attributes that can be used with this pragma directive, see *Type attributes*, page 239.

Description

Use this pragma directive to specify IAR-specific *type attributes*, which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In this example, an `int` object with the memory attribute `__brel` is defined:

```
#pragma type_attribute=__brel
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__brel int x;
```

See also

The chapter *Extended keywords* for more information.

unroll

Syntax

```
#pragma unroll=n
```

Parameters

n	The number of loop bodies in the unrolled loop, a constant integer.

Description

Use this pragma directive to specify that the loop following immediately after the directive should be unrolled and that the unrolled loop should have *n* copies of the loop body. The pragma directive can only be placed immediately before a `for`, `do`, or `while` loop, whose number of iterations can be determined at compile time.

Normally, unrolling is most effective for relatively small loops. However, in some cases, unrolling larger loops can be beneficial if it exposes opportunities for further optimizations between the unrolled loop iterations, for example common subexpression elimination or dead code elimination.

The `#pragma unroll` directive can be used to force a loop to be unrolled if the unrolling heuristics are not aggressive enough. The pragma directive can also be used to

reduce the aggressiveness of the unrolling heuristics; `#pragma unroll = 1` will prevent the unrolling of a loop.

Example

```
#pragma unroll=4
for (i = 0; i < 64; ++i)
{
  foo(i * k; (i + 1) * k);
}
```

See also

Loop unrolling, page 173.

vector

Syntax

```
#pragma vector=vector1[, vector2, vector3, ...]
```

Parameters

vectorN	The vector number(s) of an interrupt or trap function.

Description

Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example

```
#pragma vector=0x70
__interrupt void my_handler(void);
```

Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

This table summarizes the intrinsic functions:

Intrinsic function	Description
`__absolute_to_pic`	Converts a function pointer from absolute to position-independent memory position
`__code_distance`	Returns the distance between the code and the position it was linked for
`__compare_and_exchange_for_interlock`	Supports atomic lock operations using the `CAXI` instruction
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__fpu_sqrt_double`	Returns the square root of a double
`__fpu_sqrt_float`	Returns the square root of a floating-point number
`__get_interrupt_state`	Returns the interrupt state
`__get_processor_register`	Returns the value of the processor register
`__halt`	Inserts a `HALT` instruction
`__no_operation`	Inserts a `NOP` instruction
`__pic_to_absolute`	Converts a function pointer from position-independent to absolute memory position

Table 35: Intrinsic functions summary

Intrinsic function	Description
__saturated_add	Generates a saturated addition instruction
__saturated_sub	Generates a saturated subtraction instruction
__search_ones_left	Searches for the leftmost 1
__search_ones_right	Searches for the rightmost 1
__search_zeros_left	Searches for the leftmost 0
__search_zeros_right	Searches for the rightmost 0
__set_interrupt_state	Restores the interrupt state
__set_processor_register	Assigns a value to a processor register
__synchronize_exceptions	Synchronizes exceptions
__synchronize_memory	Synchronizes memory devices
__synchronize_pipeline	Synchronizes the pipeline
__upper_mul64	Returns the 32 most significant bits of a 64-bit multiplication of two 32-bit long values.

Table 35: Intrinsic functions summary (Continued)

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

__absolute_to_pic

Syntax

```
void *__absolute_to_pic(void *);
```

Description

Converts a function pointer from an absolute to a position-independent memory position. This function is only available in the position-independent code model.

In this code model, code pointers are represented as the position that the code should have had if it had not been moved. At the actual call site, the distance that the code has moved is added in order to produce the current position of the code.

The `__absolute_to_pic` intrinsic function converts a function pointer from the actual memory position to a format used in the position-independent code model.

Note that this intrinsic function returns a `void` pointer that must be explicitly converted to the appropriate type.

Example

Here we call a function that we know is positioned at address `0x10000`:

```
typedef void (* my_function_pointer_type)(void);
fp = (my_function_pointer_type) __absolute_to_pic(0x10000);
(*fp)();
```

See also

Position-independent code, page 64

__code_distance

Syntax

```
long __code_distance(void);
```

Description

Returns the number of bytes between the placement of code in memory and the position it was linked for.

This intrinsic function is only available in the position-independent code model.

See also

Position-independent code, page 64

__compare_and_exchange_for_interlock

Syntax

```
int __compare_and_exchange_for_interlock(int * ptr,
   int old_token, int new_token);
```

Description

Performs the following in an atomic operation using the assembler instruction `CAXI`: it reads the value that `ptr` points to, and if it is equal to `old_token` it is assigned the value of `new_token`.

Returns the value originally stored in the location pointed to by `ptr`.

This intrinsic function is intended for use in a concurrent environment where atomic operations are required when implementing for example a semaphore or a mutex.

Example

```
#include "intrinsics.h"

#define FREE   0
#define LOCKED 1

extern int lock;

void myFunction(void)
{
  if (__compare_and_exchange_for_interlock(&lock, FREE, LOCKED)
      == FREE)
  {
    /* Do something */

    __compare_and_exchange_for_interlock(&lock, LOCKED, FREE);
  }
}
```

__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.

__fpu_sqrt_double

Syntax

```
double __fpu_sqrt_double(double);
```

Description

Calculates the square root by using the FPU instruction `SQRTF`.

The `sqrt` function provided by the runtime library also uses the dedicated FPU instruction.

This intrinsic function is only available when the corresponding FPU is used.

Example

```
#include "intrinsics.h"

double hypot(double x, double y)
{
  return __fpu_sqrt_double(x*x, y*y);
}
```

__fpu_sqrt_float

Syntax

```
double __fpu_sqrt_float(float);
```

Description

Calculates the square root by using the FPU instruction `SQRTF`.

The `sqrt` function provided by the runtime library also uses the dedicated FPU instruction.

This intrinsic function is only available when the corresponding FPU is used.

Example

```
#include "intrinsics.h"

float hypot(float x, float y)
{
  return __fpu_sqrt_float(x*x, y*y);
}
```

__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"

/*
   This is what the __monitor keyword does.
*/
void CriticalFcn(void)
{
  __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_processor_register

Syntax

```
unsigned long __get_processor_register(int reg);
```

Description

Returns the value of the processor register *reg*. Both normal processor registers like `R8`, and special system registers can be accessed. The supported registers are defined in the header file `intrinsics.h`.

For registers `R0` to `R31`, the following aliases can be used:

Register	Alias
`Reg_R2`	`Reg_HP`
`Reg_R3`	`Reg_SP`
`Reg_R4`	`Reg_GP`
`Reg_R30`	`Reg_EP`
`Reg_R31`	`Reg_LP`

Note: This intrinsic function only works when applied to a literal constant, because the function is expanded into an `STSR` or `MOV` instruction that must know which system register it operates on at compile time.

__halt

Syntax `void __halt(void);`

Description Inserts a `HALT` instruction.

__no_operation

Syntax `void __no_operation(void);`

Description Inserts a `NOP` instruction.

__pic_to_absolute

Syntax `void *__pic_to_absolute(void *);`

Description Converts a function pointer from the representation used in the position-independent code model to where it is located in memory. This function is only available in the position-independent code model.

In the position-independent code model, code pointers are represented as the position it was originally linked for.

Note that this intrinsic function returns a void pointer that must be explicitly converted to the appropriate type.

See also *Position-independent code*, page 64

__saturated_add

Syntax `int __saturated_add(int, int);`

Description Returns the result of a saturated addition. A saturated operation computes just like a normal operation unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

For example, the maximum positive value that an `int` can store is `0x7FFFFFFF`. A normal addition between `0x7FFFFFFE` (that is, one less than the maximal positive value) and `2` will result in `0x8000000`. The result of a saturated addition will be `0x7FFFFFFF`.

__saturated_sub

Syntax

```
int __saturated_sub(int, int);
```

Description

Returns the result of a saturated subtraction. A saturated operation computes just like a normal operation unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__search_ones_left

Syntax

```
int __search_ones_left(unsigned int);
```

Description

Searches for the leftmost `1` and returns the bit position, where 1 represents the leftmost (most significant) bit and 32 the rightmost (least significant) bit. If no bit is found, `0` is returned.

This intrinsic function is implemented using the assembler instruction `SCH1L`, and is available for V850E2 and higher.

Example

See *__search_ones_right*, page 278, for a similar example.

__search_ones_right

Syntax

```
int __search_ones_right(unsigned int);
```

Description

Searches for the rightmost `1` and returns the bit position, where 1 represents the rightmost (least significant) bit and 32 the leftmost (most significant) bit. If no bit is found, `0` is returned.

This intrinsic function is implemented using the assembler instruction `SCH1R`, and is available for V850E2 and higher.

Example

```
#include "intrinsics.h"

/*
   Left adjust the value in 'x'.
   Return 0 if 'x' is zero.
*/
int left_adjust(unsigned int x)
{
  int left_zeroes  = __search_ones_left(x) - 1;
  if (left_zeroes < 0) return 0;  // x was 0
  return x << left_zeroes;
}
```

__search_zeros_left

Syntax `int __search_zeros_left(unsigned int);`

Description Searches for the leftmost `0` and returns the bit position, where 1 represents the leftmost (most significant) bit and 32 the rightmost (least significant) bit. If no bit is found, `0` is returned.

This intrinsic function is implemented using the assembler instruction `SCH0L`.

Example See *__search_ones_right*, page 278, for a similar example.

__search_zeros_right

Syntax `int __search_zeros_right(unsigned int);`

Description Searches for the rightmost `0` and returns the bit position, where 1 represents the rightmost (least significant) bit and 32 the leftmost (most significant) bit. If no bit is found, `0` is returned.

This intrinsic function is implemented using the assembler instruction `SCH0R`.

Example See *__search_ones_right*, page 278, for a similar example.

__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 275.

__set_processor_register

Syntax `void __set_processor_register(int reg, unsigned long value);`

Description Sets the register *`reg`* to the value *`value`*.

This intrinsic function can be used to set a processor register to a specific value. The registers supported are the same as for *__get_processor_register*, page 276.

__synchronize_exceptions

Syntax
```
void __synchronize_exceptions(void);
```

Description

Synchronizes exceptions before continuing execution by using the assembler instruction `SYNCE`.

__synchronize_memory

Syntax
```
void __synchronize_memory(void);
```

Description

Synchronizes memory devices before continuing execution by using the assembler instruction `SYNCM`.

__synchronize_pipeline

Syntax
```
void __synchronize_pipeline(void);
```

Description

Synchronizes the pipeline before continuing execution by using the assembler instruction `SYNCP`.

__upper_mul64

Syntax
```
long __upper_mul64(long x,long y);
```

Description

Returns the 32 most significant bits of a 64-bit multiplication of two 32-bit long values. This intrinsic function is only available on V850E and above as it is implemented by using the `MUL` instruction.

The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for V850 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 282.
- 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 200.
- 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 286.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 221.

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

See also *__FILE__*, page 283, and *--no_path_in_file_macros*, page 215.

__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 symbol reflects the `--code_model` option and is defined to `__CODE_MODEL_NORMAL__`, `__CODE_MODEL_LARGE__`, or `__CODE_MODEL_PIC__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.

__CORE__

Description

An integer that identifies the chip core in use. The symbol reflects the `--cpu` option and is defined to one of `__CORE_V850__`, `__CORE_V850E__`, `__CORE_V850E2__`, `__CORE_V850E2M__`, or `__CORE_V850E2S__`. These symbolic names can be used when testing the `__CORE__` symbol.

__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.

__CPU__

Description

Deprecated, the same as `__CORE__`.

__DATA_MODEL__

Description

An integer that identifies the data model in use. The symbol reflects the `--data_model` option and is defined to one of:

```
__DATA_MODEL_SMALL__
__DATA_MODEL_TINY__
__DATA_MODEL_TINY_WITH_SADDR__
__DATA_MODEL_SMALL__
__DATA_MODEL_SMALL_WITH_SADDR__
__DATA_MODEL_MEDIUM__
__DATA_MODEL_MEDIUM_WITH_SADDR__
__DATA_MODEL_LARGE__
__DATA_MODEL_LARGE_WITH_SADDR__
```

These symbolic names can be used when testing the `__DATA_MODEL__` symbol.

__DATE__

Description

A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2010`"

This symbol is required by Standard C.

__embedded_cplusplus

Description

An integer which is defined to `1` when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

__FILE__

Description

A string that identifies the name of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

See also

See also *__BASE_FILE__*, page 282, and *--no_path_in_file_macros*, page 215.

__FPU__

Description

Specifies the selected floating-point unit. The symbol reflects the `--fpu` option and is defined to one of `__FPU_NONE__`, `__FPU_V850E1_SINGLE__`, `__FPU_V850E2_SINGLE__`, or `__FPU_V850_DOUBLE__`.

__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 206. See also *__PRETTY_FUNCTION__*, page 285.

__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 206. See also *__PRETTY_FUNCTION__*, page 285.

__IAR_SYSTEMS_ICC__

Description

An integer that identifies the IAR compiler platform. The current value is 8. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.

__ICCV850__

Description

An integer that is set to `1` when the code is compiled with the IAR C/C++ Compiler for V850.

__LINE__

Description

An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

__LITTLE_ENDIAN__

Description

An integer that identifies the byte order of the microcontroller. For the V850 microcontroller family, the value of this symbol is defined to `1` (`TRUE`), which means that the byte order is little-endian.

__PRETTY_FUNCTION__

Description

A predefined string identifier that is initialized with the function name, including parameter types and return type, of the function in which the symbol is used, for example `"void func(char)"`. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

See also

-e, page 206. See also *__func__*, page 284.

__SADDR_ACTIVE__

Description

This predefined symbol expands to `1` if a data model with short address support is in use, otherwise it is undefined.

__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.

__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.

See also

Assert, page 112.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

#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

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

Library overview

The compiler comes with the IAR DLIB Library, 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 37 . 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`, `wcsrtomb`, `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 `printf`, `sprintf`, `scanf`, `sscanf`, `getchar`, and `putchar`.

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 V850 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 295.

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
`assert.h`	Enforcing assertions when functions execute
`complex.h`	Computing common complex mathematical functions
`ctype.h`	Classifying characters
`errno.h`	Testing error codes reported by library functions
`fenv.h`	Floating-point exception flags
`float.h`	Testing floating-point type properties

Table 36: Traditional Standard C header files—DLIB

Header file	Usage
inttypes.h	Defining formatters for all types defined in stdint.h
iso646.h	Using Amendment 1—iso646.h standard header
limits.h	Testing integer type properties
locale.h	Adapting to different cultural conventions
math.h	Computing common mathematical functions
setjmp.h	Executing non-local goto statements
signal.h	Controlling various exceptional conditions
stdarg.h	Accessing a varying number of arguments
stdbool.h	Adds support for the bool data type in C.
stddef.h	Defining several useful types and macros
stdint.h	Providing integer characteristics
stdio.h	Performing input and output
stdlib.h	Performing a variety of operations
string.h	Manipulating several kinds of strings
tgmath.h	Type-generic mathematical functions
time.h	Converting between various time and date formats
uchar.h	Unicode functionality (IAR extension to Standard C)
wchar.h	Support for wide characters
wctype.h	Classifying wide characters

Table 36: Traditional Standard C header files—DLIB (Continued)

C++ HEADER FILES

This section lists the C++ header files:

- The C++ library header files

  The header files that constitute the Embedded C++ library.
- The C++ standard template library (STL) header files

  The header files that constitute STL for the Extended Embedded C++ library.
- The C++ C header files

  The C++ header files that provide the resources from the C library.

The C++ library header files

This table lists the header files that can be used in Embedded C++:

Header file	Usage
complex	Defining a class that supports complex arithmetic
fstream	Defining several I/O stream classes that manipulate external files
iomanip	Declaring several I/O stream manipulators that take an argument
ios	Defining the class that serves as the base for many I/O streams classes
iosfwd	Declaring several I/O stream classes before they are necessarily defined
iostream	Declaring the I/O stream objects that manipulate the standard streams
istream	Defining the class that performs extractions
new	Declaring several functions that allocate and free storage
ostream	Defining the class that performs insertions
sstream	Defining several I/O stream classes that manipulate string containers
streambuf	Defining classes that buffer I/O stream operations
string	Defining a class that implements a string container
strstream	Defining several I/O stream classes that manipulate in-memory character sequences

Table 37: 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
algorithm	Defines several common operations on sequences
deque	A deque sequence container
functional	Defines several function objects
hash_map	A map associative container, based on a hash algorithm
hash_set	A set associative container, based on a hash algorithm
iterator	Defines common iterators, and operations on iterators
list	A doubly-linked list sequence container
map	A map associative container
memory	Defines facilities for managing memory
numeric	Performs generalized numeric operations on sequences

Table 38: Standard template library header files

Header file	Description
queue	A queue sequence container
set	A set associative container
slist	A singly-linked list sequence container
stack	A stack sequence container
utility	Defines several utility components
vector	A vector sequence container

Table 38: 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
cassert	Enforcing assertions when functions execute
cctype	Classifying characters
cerrno	Testing error codes reported by library functions
cfloat	Testing floating-point type properties
cinttypes	Defining formatters for all types defined in stdint.h
climits	Testing integer type properties
clocale	Adapting to different cultural conventions
cmath	Computing common mathematical functions
csetjmp	Executing non-local goto statements
csignal	Controlling various exceptional conditions
cstdarg	Accessing a varying number of arguments
cstdbool	Adds support for the bool data type in C.
cstddef	Defining several useful types and macros
cstdint	Providing integer characteristics
cstdio	Performing input and output
cstdlib	Performing a variety of operations
cstring	Manipulating several kinds of strings
ctime	Converting between various time and date formats

Table 39: New Standard C header files—DLIB

Header file	Usage
cwchar	Support for wide characters
cwctype	Classifying wide characters

Table 39: 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:

- `fenv.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

fenv.h

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapenable` and `fegettrapdisable`.

stdio.h

These functions provide additional I/O functionality:

`fdopen`	Opens a file based on a low-level file descriptor.
`fileno`	Gets the low-level file descriptor from the file descriptor (`FILE*`).
`__gets`	Corresponds to `fgets` on `stdin`.
`getw`	Gets a `wchar_t` character from `stdin`.
`putw`	Puts a `wchar_t` character to `stdout`.
`__ungetchar`	Corresponds to `ungetc` on `stdout`.
`__write_array`	Corresponds to `fwrite` on `stdout`.

string.h

These are the additional functions defined in `string.h`:

`strdup`	Duplicates a string on the heap.
`strcasecmp`	Compares strings case-insensitive.
`strncasecmp`	Compares strings case-insensitive and bounded.
`strnlen`	Bounded string length.

time.h

There are two interfaces for using `time_t` and the associated functions `time`, `ctime`, `difftime`, `gmtime`, `localtime`, and `mktime`:

- The 32-bit interface supports years from 1900 up to 2035 and uses a 32-bit integer for `time_t`. The type and function have names like `__time32_t`, `__time32`, etc. This variant is mainly available for backwards compatibility.
- The 64-bit interface supports years from -9999 up to 9999 and uses a signed `long long` for `time_t`. The type and function have names like `__time64_t`, `__time64`, etc.

The interfaces are defined in 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 109.

SYMBOLS USED INTERNALLY BY THE LIBRARY

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__code`

This symbol is used as a memory attribute internally by the compiler, and it might have to be used as an argument in certain templates.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor`, `__has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

Note: The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

Segment reference

The compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For more information about segments, see the chapter *Placing code and data*.

Summary of segments

The table below lists the segments that are available in the compiler:

Segment	Description
BREL_BASE	An empty placeholder segment, used as a base for the BREL and BREL23 segments holding variable data.
BREL_CBASE	An empty placeholder segment, used as a base for the BREL and BREL23 segment holding constant data.
BREL_C	Holds __brel constant data.
BREL_I	Holds __brel static and global initialized variables.
BREL_ID	Holds initial values for __brel static and global variables in BREL_I.
BREL_N	Holds __no_init __brel static and global variables.
BREL_Z	Holds zero-initialized __brel static and global variables.
BREL23_C	Holds __brel23 constant data.
BREL23_I	Holds __brel23 static and global initialized variables.
BREL23_ID	Holds initial values for __brel23 static and global variables in BREL23_I.
BREL23_N	Holds __no_init __brel23 static and global variables.
BREL23_Z	Holds zero-initialized __brel23 static and global variables.
CHECKSUM	Holds the checksum generated by the linker.
CLTCODE	Holds code for the callt functions.
CLTVEC	Holds the vector of callt functions.
CODE	Holds the program code.
CSTACK	Holds the stack used by C or C++ programs.

Table 40: Segment summary

Segment	Description
CSTART	Holds the startup code.
DIFUNCT	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before main is called.
GLOBAL_AC	Holds located constant data.
GLOBAL_AN	Holds located uninitialized data.
HEAP	Holds the heap used for dynamically allocated data.
HUGE_C	Holds __huge constant data.
HUGE_I	Holds __huge static and global initialized variables.
HUGE_ID	Holds initial values for __huge static and global variables in HUGE_I.
HUGE_N	Holds __no_init __huge static and global variables.
HUGE_Z	Holds zero-initialized __huge static and global variables.
ICODE	Holds the interrupt code.
INTVEC	Contains the reset and interrupt vectors.
NEAR_C	Holds __near constant data.
NEAR_I	Holds __near static and global initialized variables.
NEAR_ID	Holds initial values for __near static and global variables in NEAR_I.
NEAR_N	Holds __no_init __near static and global variables.
NEAR_Z	Holds zero-initialized __near static and global variables.
RCODE	Holds the library code.
SADDR_BASE	An empty placeholder segment, used as a base for the SADDR7 and SADDR8 segment groups.
SADDR7_I	Holds __saddr static and global initialized character variables.
SADDR7_ID	Holds initial values for __saddr static and global character variables in SADDR7_I.
SADDR7_N	Holds __no_init __saddr static and global character variables.
SADDR7_Z	Holds zero-initialized __saddr static and global character variables.
SADDR8_I	Holds __saddr static and global initialized non-character variables.
SADDR8_ID	Holds initial values for __saddr static and global non-character variables in SADDR8_I.
SADDR8_N	Holds __no_init __saddr static and global non-character variables.
SADDR8_Z	Holds zero-initialized __saddr static and global non-character variables.
SYSCALLCODE	Holds code for the syscsall functions.

Table 40: Segment summary (Continued)

Segment	Description
SYSCALLVEC	Holds the vector of syscall functions.
TRAPVEC	Holds the trap vector.

Table 40: 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.

In each description, the segment memory type—CODE, CONST, or DATA—indicates whether the segment should be placed in ROM or RAM memory; see Table 7, *XLINK segment memory types*, page 68.

For information about the -Z and the -P directives, see the *IAR Linker and Library Tools Reference Guide.*

For information about how to define segments in the linker configuration file, see *Customizing the linker configuration file*, page 68.

For more information about the extended keywords mentioned here, see the chapter *Extended keywords*.

BREL_BASE

Description

Empty placeholder segment; a base for the BREL and BREL23 segments holding data. The brel base register GP (register R4) points with an 0x8000 offset to this segment. The segments of the BREL segment group holding RAM data—BREL_I, BREL_N, and BREL_Z—must be placed within 64 Kbytes following the placeholder segment. The segments of the BREL23 segment group holding RAM data must be placed within ±4 Mbytes of the base pointer, which points to BREL_BASE + 0x8000.

Segment memory type

DATA

Memory placement

This segment can be placed anywhere in memory.

BREL_CBASE

Description

Empty placeholder segment; a base for the BREL and BREL23 segments holding *constant* data. The brel constant base register (R25) points with an 0x8000 offset to this segment.

The segment BREL_C must be placed within 64 Kbytes following the placeholder segment. The segment BREL23_C must be placed ±4 Mbytes of the base pointer, which points to BREL_CBASE + 0x8000.

Segment memory type CONST

Memory placement This segment can be placed anywhere in memory.

BREL_C

Description Holds __brel constant data. This can include constant variables, string and aggregate literals, etc.

Segment memory type CONST

Memory placement This segment must be placed within 64 Kbytes following BREL_CBASE.

Access type Read-only

BREL_I

Description Holds __brel static and global initialized variables initialized by copying from the segment BREL_ID at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type DATA

Memory placement This segment must be placed within 64 Kbytes following BREL_BASE.

Access type Read-write

BREL_ID

Description Holds initial values for __brel static and global variables in the BREL_I segment. These values are copied from BREL_ID to BREL_I at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type CONST

Memory placement This segment can be placed anywhere in memory.

Access type Read-only

BREL_N

Description Holds static and global __no_init __brel variables.

Segment memory type DATA

Memory placement This segment must be placed within 64 Kbytes following BREL_BASE.

Access type Read-write

BREL_Z

Description Holds zero-initialized __brel static and global variables. The contents of this segment is declared by the system startup code.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type DATA

Memory placement This segment must be placed within 64 Kbytes following BREL_BASE.

Access type Read-write

BREL23_C

Description Holds __brel23 constant data. This can include constant variables, string and aggregate literals, etc. This segment is only used for V850E2M and above.

Segment memory type CONST

Memory placement This segment must be placed within ±4 Mbytes around the constant brel base pointer (R25) which points to BREL_CBASE + 0x8000.

Access type Read-only

BREL23_I

Description: Holds __brel23 static and global initialized variables initialized by copying from the segment BREL23_ID at application startup. This segment is only used for V850E2M and above.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type: DATA

Memory placement: This segment must be placed within ±4 Mbytes around the brel base pointer (GP) which points to BREL_BASE + 0x8000.

Access type: Read-write

BREL23_ID

Description: Holds initial values for __brel23 static and global variables in the BREL23_I segment. These values are copied from BREL23_ID to BREL23_I at application startup. This segment is only used for V850E2M and above.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type: CONST

Memory placement: This segment can be placed anywhere in memory.

Access type: Read-only

BREL23_N

Description: Holds static and global __no_init __brel23 variables. This segment is only used for V850E2M and above.

Segment memory type: DATA

Memory placement
This segment must be placed within ±4 Mbytes around the brel base pointer (GP) which points to BREL_BASE + 0x8000.

Access type
Read-write

BREL23_Z

Description
Holds zero-initialized __brel23 static and global variables. The contents of this segment is declared by the system startup code. This segment is only used for V850E2M and above.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type
DATA

Memory placement
This segment must be placed within ±4 Mbytes around the brel base pointer (GP) which points to BREL_BASE + 0x8000.

Access type
Read-write

CHECKSUM

Description
Holds the checksum bytes generated by the linker. This segment also holds the __checksum symbol. Note that the size of this segment is affected by the linker option -J.

Segment memory type
CONST

Memory placement
This segment can be placed anywhere in ROM memory.

Access type
Read-only

CLTCODE

Description
Holds code for callt functions. This segment must be placed following the CLTVEC segment.

Segment memory type
CODE

Memory placement — This segment must be placed within 64 Kbytes following CLTVEC, and in the same 2-Mbyte area as CODE if normal functions are called from callt functions.

Access type — Read-only

CLTVEC

Description — Holds the callt vector table generated by the use of the extended keyword __callt.

Segment memory type — CONST

Memory placement — This segment can be placed anywhere in memory.

Access type — Read-only

CODE

Description — Holds program code, except the code for system initialization.

Segment memory type — CODE

Memory placement — In the Normal and Position-independent code models, this segment must be placed within a 2-Mbyte area. The area must also include the CSTART segment. If any calls are performed from interrupt or callt functions, the corresponding segment, ICODE or CLTCODE, respectively, must be placed in this 2-Mbyte area. Function calls can originate from this segment or from the CLTCODE, ICODE, and CSTART segments.

In the Large code model, 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 — *The stack*, page 74.

CSTART

Description — Holds the startup code.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type — `CODE`

Memory placement — This segment must be placed in the same 2-Mbyte area as `CODE`.

Access type — Read-only

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

GLOBAL_AC

Description — Holds located constant data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker configuration file.

GLOBAL_AN

Description — Holds `__no_init` located data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker configuration file.

HEAP

Description: Holds the heap used for dynamically allocated data, in other words data allocated by `malloc` and `free`, and in C++, `new` and `delete`.

Segment memory type: `DATA`

Memory placement: This segment can be placed anywhere in memory.

Access type: Read-write

See also: *The heap*, page 75.

HUGE_C

Description: Holds `__huge` 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

HUGE_I

Description: Holds `__huge` static and global initialized variables initialized by copying from the segment `HUGE_ID` at application startup.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type: `DATA`

Memory placement: This segment can be placed anywhere in memory.

Access type: Read-write

HUGE_ID

Description	Holds initial values for `__huge` static and global variables in the `HUGE_I` segment. These values are copied from `HUGE_ID` to `HUGE_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`CONST`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

HUGE_N

Description	Holds static and global `__no_init __huge` variables.
Segment memory type	`DATA`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-write

HUGE_Z

Description	Holds zero-initialized `__huge` static and global variables. The contents of this segment is declared by the system startup code. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-write

ICODE

Description: Holds interrupt code. Must be able to be called from the interrupt vector table.

Segment memory type: `CODE`

Memory placement: This segment must be placed in one of the ranges `0x0–0x1FFFFF` and `0xFFE00000–0xFFFFFFFF`. This segment must also be located in the same 2-Mbyte area as `CODE` if normal functions are called from interrupt functions.

Access type: Read-only

INTVEC

Description: Holds the interrupt vector table generated by the use of the `__interrupt` and `__trap` extended keywords in combination with the `#pragma vector` directive.

Segment memory type: `CODE`

Memory placement: This segment must be placed at address `0x0`.

Access type: Read-only

NEAR_C

Description: Holds `__near` constant data. This can include constant variables, string and aggregate literals, etc.

Segment memory type: `CONST`

Memory placement: This segment must be placed in one of the ranges `0x0–0x7FFF` and `0xFFFF8000–0xFFFFFFFF`.

Access type: Read-only

NEAR_I

Description: Holds `__near` static and global initialized variables initialized by copying from the segment `NEAR_ID` at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type DATA

Memory placement This segment must be placed in one of the ranges 0x0–0x7FFF and 0xFFFF8000–0xFFFFFFFF.

Access type Read-write

NEAR_ID

Description Holds initial values for __near static and global variables in the NEAR_I segment. These values are copied from NEAR_ID to NEAR_I at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type CONST

Memory placement This segment can be placed anywhere in memory.

Access type Read-only

NEAR_N

Description Holds static and global __no_init __near variables.

Segment memory type DATA

Memory placement This segment must be placed in one of the ranges 0x0–0x7FFF and 0xFFFF8000–0xFFFFFFFF.

Access type Read-write

NEAR_Z

Description Holds zero-initialized __near static and global variables. The contents of this segment is declared by the system startup code.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type DATA

Memory placement This segment must be placed in one of the ranges 0x0–0x7FFF and 0xFFFF8000–0xFFFFFFFF.

Access type Read-write

RCODE

Description Holds library code.

Segment memory type CODE

Memory placement This segment must be placed in the same 2-Mbyte area as the CODE segment.

Access type Read-only

SADDR_BASE

Description Empty placeholder segment, a base for the SADDR7 and SADDR8 segment groups. The saddr base pointer will point to this segment. The segments of the SADDR7 and SADDR8 segment groups (with the exception of SADDR7_ID and SADDR8_ID) must be placed within 128 and 256 bytes, respectively, following this segment.

Segment memory type Any

Memory placement This segment can be placed anywhere in memory.

SADDR7_I

Description Holds __saddr static and global initialized character variables with a 128-byte offset, initialized by copying from the segment SADDR7_ID at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type DATA

Memory placement This segment must be placed within 128 bytes following SADDR_BASE.

Access type Read-write

SADDR7_ID

Description Holds initial values for __saddr static and global character variables with a 128-byte offset in the SADDR7_I segment. These values are copied from SADDR7_ID to SADDR7_I at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type CONST

Memory placement This segment can be placed anywhere in memory.

Access type Read-only

SADDR7_N

Description Holds static and global __no_init __saddr character variables with a 128-byte offset.

Segment memory type DATA

Memory placement This segment must be placed within 128 bytes following SADDR_BASE.

Access type Read-write

SADDR7_Z

Description Holds zero-initialized __saddr static and global character variables with a 128-byte offset. The contents of this segment is declared by the system startup code.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type DATA

Memory placement — This segment must be placed within 128 bytes following SADDR_BASE.

Access type — Read-write

SADDR8_I

Description — Holds __saddr static and global initialized non-character variables with a 256-byte offset, initialized by copying from the segment SADDR8_ID at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type — DATA

Memory placement — This segment must be placed within 256 bytes following SADDR_BASE.

Access type — Read-write

SADDR8_ID

Description — Holds initial values for __saddr static and global non-character variables with a 256-byte offset in the SADDR8_I segment. These values are copied from SADDR8_ID to SADDR8_I at application startup.

This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type — CONST

Memory placement — This segment can be placed anywhere in memory.

Access type — Read-only

SADDR8_N

Description — Holds static and global __no_init __saddr non-character variables with a 256-byte offset.

Segment memory type — DATA

Memory placement — This segment must be placed within 256 bytes following `SADDR_BASE`.

Access type — Read-write

SADDR8_Z

Description — Holds zero-initialized `__saddr` static and global non-character variables with a 256-byte offset. The contents of this segment is declared by the system startup code.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type — `DATA`

Memory placement — This segment must be placed within 256 bytes following `SADDR_BASE`.

Access type — Read-write

SYSCALLCODE

Description — Holds code for syscall functions.

Segment memory type — `CODE`

Memory placement — This segment can be placed anywhere in memory.

Access type — Read-only

SYSCALLVEC

Description — Holds the callt vector table generated by the use of the extended keyword `__syscall`.

Segment memory type — `CONST`

Memory placement — This segment can be placed anywhere in memory.

Access type — Read-only

TRAPVEC

Description	Holds the trap vector table.
Segment memory type	CONST
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

Implementation-defined behavior for Standard C

This chapter describes how the compiler handles the implementation-defined areas of the C language based on Standard C.

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.

If you are using C89 instead of Standard C, see *Implementation-defined behavior for C89*, page 333. For a short overview of the differences between Standard C and C89, see *C language overview*, page 145.

Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

J.3.1 TRANSLATION

Diagnostics (3.10, 5.1.1.3)

Diagnostics are produced in the form:

filename,*linenumber level*[*tag*]: *message*

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.

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 99.

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 105.

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*, page 228.

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 233.

ptrdiff_t (6.5.6)

For information about `ptrdiff_t`, see *ptrdiff_t*, page 233.

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 63.

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 229.

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 206.

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 229.

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 227.

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 235.

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 281.

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 198.

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 186.

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 186.

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 *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter 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
public_equ
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—requiere a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 81.

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 295.

Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 231.

Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 105.

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 109.

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 105.

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 105.

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 105.

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 109.

Range and precision of time (7.23)

The implementation uses `signed long` for representing `clock_t` and `time_t`, based at the start of the year 1970. This gives a range of approximately plus or minus 69 years in seconds. However, the application must supply the actual implementation for the functions `time` and `clock`. See *Time*, page 109.

clock() (7.23.2.1)

The application must supply an implementation of the `clock` function. See *Time*, page 109.

%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 109.

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 227.

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 227.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 227.

J.4 LOCALE

Members of the source and execution character set (5.2.1)

By default, the compiler accepts all one-byte characters in the host's default character set. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

The meaning of the additional character set (5.2.1.2)

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

Shift states for encoding multibyte characters (5.2.1.2)

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

Direction of successive printing characters (5.2.2)

The application defines the characteristics of a display device.

The decimal point character (7.1.1)

The default decimal-point character is a '`.`'. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

Printing characters (7.4, 7.25.2)

The set of printing characters is determined by the chosen locale.

Control characters (7.4, 7.25.2)

The set of control characters is determined by the chosen locale.

Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

The sets of characters tested are determined by the chosen locale.

The native environment (7.1.1.1)

The native environment is the same as the "C" locale.

Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

The collation of the execution character set (7.21.4.3, 7.24.4.4.2)

The collation of the execution character set is determined by the chosen locale.

Message returned by strerror (7.21.6.2)

The messages returned by the `strerror` function depending on the argument is:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`EFPOS`	`file positioning error`
`EILSEQ`	`multi-byte encoding error`
`<0 || >99`	`unknown error`
all others	`error` *nnn*

Table 41: Message returned by strerror()—IAR DLIB library

Implementation-defined behavior for C89

This chapter describes how the compiler handles the implementation-defined areas of the C language based on the C89 standard.

If you are using Standard C instead of C89, see *Implementation-defined behavior for Standard C*, page 317. For a short overview of the differences between Standard C and C89, see *C language overview*, page 145.

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 (5.1.2.2.2.1)

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 99.

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 105.

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 105.

Range of 'plain' char (6.2.1.1)

A 'plain' `char` has the same range as an `unsigned char`.

INTEGERS

Range of integer values (6.1.2.5)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types*, page 228, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (`s`), a biased exponent (`e`), and a mantissa (`m`).

See *Floating-point types*, page 231, 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 233, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 233, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 233, 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*, page 228, 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 *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter 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`

`public_equ`

`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 109.

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 105.

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 105.

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 105.

%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 108.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 108.

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` *nnn*

Table 42: 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 109.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 109.

A

abort
- implementation-defined behavior. 329
- implementation-defined behavior in C89 (DLIB) 342
- system termination (DLIB) 98

absolute location
- data, placing at (@) 168
- language support for 148
- #pragma location 259

__absolute_to_pic (intrinsic function). 65, 272
addressing. *See* memory types, data models, and code models
--aggressive_inlining (compiler option) 197
--aggressive_unrolling (compiler option) 197
algorithm (STL header file) 293
alignment 227
- forcing stricter (#pragma data_alignment) 254
- in structures (#pragma pack) 263
- in structures, causing problems 164
- of an object (__ALIGNOF__) 148
- of data types. 227
- restrictions for inline assembler 120

alignment (pragma directive) 324, 339
__ALIGNOF__ (operator) 148
--allow_misaligned_data_access (compiler option) 198
anonymous structures 165
anonymous symbols, creating 145
ANSI C. *See* C89
application
- building, overview of 36
- startup and termination (DLIB) 95

architecture
- more information about 25
- of V850 43

ARGFRAME (assembler directive) 131
argv (argument), implementation-defined behavior 318
arrays
- designated initializers in 145
- global, accessing 135
- implementation-defined behavior. 322
- implementation-defined behavior in C89 336
- incomplete at end of structs 145
- non-lvalue 151
- of incomplete types 150
- single-value initialization. 151

asm, __asm (language extension) 119
assembler code
- calling from C 121
- calling from C++ 123
- inserting inline 119

assembler directives
- for call frame information 140
- for static overlay 131
- using in inline assembler code 120

assembler instructions
- CAXI 273
- CLR1 134
- inserting inline 119
- LD 134, 138
- MOVHI 138
- NOT1 134
- SCH0L. 279
- SCH0R. 279
- SCH1L. 278
- SCH1R. 278
- SET1 134
- SLD 134
- SST 134
- ST 134, 138
- SYNCE 280
- SYNCM. 280
- SYNCP 280
- TST1 134
- used for calling functions. 131

assembler labels
- ?BREL_BASE 137
- ?BREL_CBASE. 137

?SADDR_BASE 139
assembler labels, making public (--public_equ) 222
assembler language interface 117
calling convention. *See* assembler code
labels 132
assembler list file, generating 209
assembler output file. 122
assembler variable, ?CODE_DISTANCE 134
asserts 112
implementation-defined behavior of 326
implementation-defined behavior of in C89, (DLIB) . . 340
including in application 286
assert.h (DLIB header file) 291
__assignment_by_bitwise_copy_allowed, symbol used
in library 296
@ (operator)
placing at absolute address. 168
placing in segments 169
atomic operations 59
__monitor 246
attributes
object 241
type 239
auto variables 50–51
at function entrance 126
programming hints for efficient code 177
using in inline assembler code 120

B

backtrace information *See* call frame information
Barr, Michael 28
baseaddr (pragma directive) 324, 339
__BASE_FILE__ (predefined symbol). 282
base-relative memory 46
accessing using assembler 137
base-relative23 memory, accessing using assembler 138
basic type names, using in preprocessor expressions
(--migration_preprocessor_extensions). 212
basic_template_matching (pragma directive) 324, 339
batch files
error return codes. 188
none for building library from command line 94
binary streams. 327
binary streams in C89 (DLIB). 341
bit negation 179
bitfields
data representation of. 229
hints 164
implementation-defined behavior. 323
implementation-defined behavior in C89 337
non-standard types in 148
bitfields (pragma directive) 253
bits in a byte, implementation-defined behavior 319
bold style, in this guide 30
bool (data type). 228
adding support for in DLIB 292, 294
__brel (extended keyword) 243
brel base pointer register, considerations. 126
brel memory. *See* base-relative memory
brel (memory type) 46
BREL_BASE (segment) 301
BREL_C (segment). 302
BREL_CBASE (segment). 301
BREL_I (segment) 302
BREL_ID (segment). 302
BREL_N (segment) 303
BREL_Z (segment). 303
__brel23 (extended keyword) 244
brel23 (memory type) 47
BREL23_C (segment). 303
BREL23_I (segment) 304
BREL23_ID (segment) 304
BREL23_N (segment) 304
BREL23_Z (segment). 305
building_runtime (pragma directive). 324, 339
__BUILD_NUMBER__ (predefined symbol) 282

byte order
identifying 285

C

C and C++ linkage 124
C/C++ calling convention. *See* calling convention
C header files 291
C language, overview 145
call frame information 139
in assembler list file 122
in assembler list file (-lA) 210
call stack 139
call table 244
callee-save registers, stored on stack 51
calling convention
C++, requiring C linkage 123
in compiler 124
calloc (library function) 52
See also heap
implementation-defined behavior in C89 (DLIB) 342
callt 97
__callt (extended keyword) 58–59, 244
callt functions
in C language 58
placement in memory 78
callt vector table, CLTVEC segment 306
callt (function type) 58
can_instantiate (pragma directive) 324, 339
cassert (library header file) 294
cast operators
in Extended EC++ 154, 158
missing from Embedded C++ 154
casting
of pointers and integers 233
pointers to integers, language extension 150
CAXI (assembler instruction) 273
cctype (DLIB header file) 294
cerrno (DLIB header file) 294
cexit (system termination code)
in DLIB 96
placement in segment 77
CFI (assembler directive) 139
CFI_COMMON (call frame information macro) 143
CFI_NAMES (call frame information macro) 143
cfi.h (CFI header example file) 140
cfi.m85 (CFI header example file) 143
cfloat (DLIB header file) 294
char (data type) 228
changing default representation (--char_is_signed) . . . 198
changing representation (--char_is_unsigned) 199
implementation-defined behavior 319
signed and unsigned 229
character set, implementation-defined behavior 318
characters, implementation-defined behavior 319
characters, implementation-defined behavior in C89 334
character-based I/O
in DLIB 101
--char_is_signed (compiler option) 198
--char_is_unsigned (compiler option) 199
CHECKSUM (segment) 305
cinttypes (DLIB header file) 294
climits (DLIB header file) 294
clocale (DLIB header file) 294
clock (DLIB library function),
implementation-defined behavior in C89 343
clock (library function)
implementation-defined behavior 330
clock.c 109
__close (DLIB library function) 105
CLR1 (assembler instruction) 134
CLTCODE (segment) 78, 305
CLTVEC (segment) 78, 306
clustering (compiler transformation) 175
disabling (--no_clustering) 213
cmath (DLIB header file) 294
code
execution of 38
interruption of execution 55

position-independent 64
verifying linked result 79
code models 53
calling functions in. 131
configuration 38
identifying (__CODE_MODEL__) 282
large 54
normal 54
position-independent 54
specifying on command line (--code_model) 199
code motion (compiler transformation). 174
disabling (--no_code_motion) 214
code segments, used for placement 77
code span, extending. 178
CODE (segment) 306
using 77
codeseg (pragma directive) 325, 339
__code_distance (intrinsic function) 65, 273
__CODE_MODEL__ (predefined symbol). 282
__code_model (runtime model attribute) 114
--code_model (compiler option) 199
__code, symbol used in library 296
command line options
See also compiler options
part of compiler invocation syntax 185
passing 185
typographic convention 30
command prompt icon, in this guide 30
comments
after preprocessor directives. 151
C++ style, using in C code 145
common block (call frame information) 140
common subexpr elimination (compiler transformation) . 173
disabling (--no_cse) 214
__compare_and_exchange_for_interlock
(intrinsic function) 273
compilation date
exact time of (__TIME__) 286
identifying (__DATE__) 283
compiler
environment variables 186
invocation syntax 185
output from 187
compiler listing, generating (-l). 209
compiler object file 36
including debug information in (--debug, -r) 201
output from compiler 187
compiler optimization levels 172
compiler options 191
passing to compiler 185
reading from file (-f) 208
specifying parameters 193
summary 193
syntax. 191
for creating skeleton code 122
instruction scheduling 176
--lock_regs 176
--lock_regs_compatibility 177
--reg_const 176
--warnings_affect_exit_code 188
compiler platform, identifying 284
compiler subversion number 286
compiler transformations 170
compiler version number 286
compiling
from the command line 36
syntax. 185
complex numbers, supported in Embedded C++. 154
complex (library header file). 293
complex.h (library header file) 291
compound literals 145
computer style, typographic convention 30
configuration
basic project settings 37
__low_level_init 99
configuration symbols
for file input and output 105
for locale 106

for printf and scanf. 103
for strtod 110
in library configuration files. 95, 100
consistency, module 112
const
declaring objects 237
non-top level 151
constants without constructors, C++ 46
constants, placing in named segment 254
__constrange(), symbol used in library 297
__construction_by_bitwise_copy_allowed, symbol used in library 297
constseg (pragma directive) 253
const_cast (cast operator) 154
contents, of this guide 25
control characters, implementation-defined behavior 331
conventions, used in this guide 29
copyright notice 2
__CORE__ (predefined symbol). 282
core
identifying 282–283
core setting 38
cos (library function) 290
cos (library routine) 110–111
cosf (library routine). 111–112
cosl (library routine) 111–112
__cplusplus (predefined symbol) 282
__CPU__ (predefined symbol) 283
__cpu (runtime model attribute) 114
cross call (compiler transformation) 175
csetjmp (DLIB header file) 294
csignal (DLIB header file) 294
cspy_support (pragma directive) 325, 339
CSTACK (segment) 306
example 74
See also stack
CSTART (segment). 77, 307
cstartup (system startup code)
code segment for 77
customizing system initialization 99
source files for (DLIB). 96
cstdarg (DLIB header file) 294
cstdbool (DLIB header file) 294
cstddef (DLIB header file) 294
cstdio (DLIB header file) 294
cstdlib (DLIB header file). 294
cstring (DLIB header file). 294
ctime (DLIB header file). 294
ctype.h (library header file). 291
cwctype.h (library header file) 295
C_INCLUDE (environment variable) 186
C-SPY
debug support for C++. 157
including debugging support 89
interface to system termination 99
Terminal I/O window, including debug support for 91
C++
See also Embedded C++ *and* Extended Embedded C++
absolute location 169
calling convention 123
dynamic initialization in 78
header files. 292
language extensions 159
special function types. 62
standard template library (STL) 293
static member variables 169
support for 35
C++ header files 293
C++ names, in assembler code 123
C++ objects, placing in memory type 50
C++ terminology. 29
C++-style comments. 145
C89
implementation-defined behavior. 333
support for 145
--c89 (compiler option). 198
C99. *See* Standard C

D

-D (compiler option) 200
data
  alignment of 227
  different ways of storing 43
  located, declaring extern 169
  placing 167, 254, 299
    at absolute location 168
  representation of 227
  storage 43
  verifying linked result 79
data block (call frame information) 140
data memory attributes, using 47
data models 44
  configuration 38
  identifying (__DATA_MODEL__) 283
data pointers 232
data segments 71
data types 228
  floating point 231
  in C++ 237
  integer types 228
dataseg (pragma directive) 254
data_alignment (pragma directive) 254
__DATA_MODEL__ (predefined symbol) 283
--data_model (compiler option) 200
__data_model (runtime model attribute) 114
__DATE__ (predefined symbol) 283
date (library function), configuring support for 109
DC32 (assembler directive) 120
--debug (compiler option) 201
debug information, including in object file 201
decimal point, implementation-defined behavior 331
declarations
  empty 151
  in for loops 145
  Kernighan & Ritchie 179
  of functions 124
declarations and statements, mixing 145
declarators, implementation-defined behavior in C89 338
define_type_info (pragma directive) 325, 339
delete (keyword) 52
denormalized numbers. *See* subnormal numbers
--dependencies (compiler option) 201
deque (STL header file) 293
destructors and interrupts, using 157
DI (assembler instruction) 274
diagnostic messages 189
  classifying as compilation errors 202
  classifying as compilation remarks 203
  classifying as compiler warnings 204
  disabling compiler warnings 218
  disabling wrapping of in compiler 218
  enabling compiler remarks 223
  listing all used by compiler 204
  suppressing in compiler 203
--diagnostics_tables (compiler option) 204
diagnostics, implementation-defined behavior 317
diag_default (pragma directive) 255
--diag_error (compiler option) 202
diag_error (pragma directive) 255
--diag_remark (compiler option) 203
diag_remark (pragma directive) 256
--diag_suppress (compiler option) 203
diag_suppress (pragma directive) 256
--diag_warning (compiler option) 204
diag_warning (pragma directive) 256
DIFUNCT (segment) 79, 307
directives
  function for static overlay 131
  pragma 41, 251
directory, specifying as parameter 192
__disable_interrupt (intrinsic function) 274
--disable_sld_suppression (compiler option) 204
--discard_unused_publics (compiler option) 205
disclaimer 2

DLIB. 39, 291
configurations 100
configuring. 82, 205
documentation 27
including debug support. 89
reference information. *See* the online help system 289
runtime environment 81
--dlib_config (compiler option). 205
DLib_Defaults.h (library configuration file) 95, 100
__DLIB_FILE_DESCRIPTOR (configuration symbol) . . 105
document conventions. 29
documentation
overview of guides 27
domain errors, implementation-defined behavior 326
domain errors, implementation-defined behavior in C89 (DLIB) 340
double (data type) 231
do_not_instantiate (pragma directive) 325, 339
dynamic initialization 95
and C++ 78
dynamic memory 52

E

-e (compiler option) 206
early_initialization (pragma directive) 325, 339
--ec++ (compiler option). 206
edition, of this guide 2
--eec++ (compiler option) 206
EI (assembler instruction) 274
Embedded C++. 153
differences from C++. 153
enabling 206
function linkage 124
language extensions 153
overview 153
Embedded C++ Technical Committee 29
embedded systems, IAR special support for 40
__embedded_cplusplus (predefined symbol) 283
__enable_interrupt (intrinsic function) 274
--enable_multibytes (compiler option) 207
entry label, program 96
enumerations, implementation-defined behavior. 323
enumerations, implementation-defined behavior in C89 . . 337
enums
data representation 228
forward declarations of 150
environment
implementation-defined behavior. 318
implementation-defined behavior in C89 333
runtime (DLIB) 81
environment names, implementation-defined behavior . . . 319
environment variables
C_INCLUDE. 186
environment (native), implementation-defined behavior 331
EP (processor register) 139
EQU (assembler directive) 222
ERANGE 326
ERANGE (C89) 340
errno value at underflow, implementation-defined behavior 329
errno.h (library header file) 291
error messages 190
classifying for compiler 202
error return codes 188
error (pragma directive) 257
--error_limit (compiler option) 207
escape sequences, implementation-defined behavior 319
exception handling, missing from Embedded C++ 153
exception vectors 77
EXCP (assembler instruction). 57–58
_Exit (library function) 98
exit (library function) 98
implementation-defined behavior. 329
implementation-defined behavior in C89 342
_exit (library function) 98
__exit (library function) 98
exp (library routine) 110

expf (library routine). 111
expl (library routine). 111
export keyword, missing from Extended EC++ 157
extended command line file
for compiler 208
passing options. 185
Extended Embedded C++ 154
enabling 206
extended keywords 239
enabling (-e). 206
overview 41
summary 242
syntax. 48
object attributes. 242
type attributes on data objects 240
type attributes on functions 241
__brel. 243
__brel23. 244
__callt 58–59, 244
See also CLTVEC (segment)
__flat 244
__huge 245
__interrupt
See also INTVEC (segment)
__near 246
__no_bit_access. 247
__saddr 248
__syscall 248
See also SYSCALLVEC (segment)
__trap
See also INTVEC (segment)
extern "C" linkage. 156

F

-f (compiler option). 208
fatal error messages 190
fdopen, in stdio.h 295
fegettrapdisable. 295
fegettrapenable 295
FENV_ACCESS, implementation-defined behavior. 322
fenv.h (library header file). 291
additional C functionality. 295
fgetpos (library function), implementation-defined behavior 329
fgetpos (library function), implementation-defined behavior in C89. 342
__FILE__ (predefined symbol). 283
file buffering, implementation-defined behavior 327
file dependencies, tracking 201
file paths, specifying for #include files 209
file position, implementation-defined behavior. 327
file (zero-length), implementation-defined behavior 327
filename
extension for linker configuration file 68
of object file. 220
search procedure for. 186
specifying as parameter 192
filenames (legal), implementation-defined behavior 327
fileno, in stdio.h 295
files, implementation-defined behavior
handling of temporary 328
multibyte characters in. 328
opening 328
__flat (extended keyword) 244
float (data type). 231
floating-point constants
hexadecimal notation 145
hints 164
floating-point environment, accessing or not 267
floating-point expressions
contracting or not. 267
floating-point expressions, using in preprocessor extensions. 212
floating-point format. 231
hints 163–164
implementation-defined behavior. 321
implementation-defined behavior in C89 336
special cases. 232

32-bits 231
64-bits 231
floating-point status flags 295
floating-point unit
identifying 284
float.h (library header file) 291
FLT_EVAL_METHOD, implementation-defined behavior 321, 326, 330
FLT_ROUNDS, implementation-defined behavior 321, 330
fmod (library function), implementation-defined behavior in C89 340
for loops, declarations in. 145
formats
floating-point values 231
standard IEEE (floating point) 231
__FPU__ (predefined symbol) 284
--fpu (compiler option) 208
FPU instructions
SQRTF. 274–275
__fpu_double (runtime model attribute) 114
__fpu_single (runtime model attribute). 114
__fpu_sqrt_double (intrinsic function) 274
__fpu_sqrt_float (intrinsic function) 275
FP_CONTRACT, implementation-defined behavior. 322
fragmentation, of heap memory 52
free (library function). *See also* heap 52
fsetpos (library function), implementation-defined behavior 329
fstream (library header file) 293
ftell (library function), implementation-defined behavior . 329
ftell (library function), implementation-defined behavior in C89 342
Full DLIB (library configuration) 100
__func__ (predefined symbol) 152, 284
FUNCALL (assembler directive) 131
__FUNCTION__ (predefined symbol) 152, 284
function calls
calling convention 124
cost of 54
large code model 133
normal code model. 132
position-independent code model. 133
stack image after 127
function calls, cost of 54
function declarations, Kernighan & Ritchie 179
function directives for static overlay 131
function inlining (compiler transformation) 174
disabling (--no_inline) 215
function pointers. 232
converting to an absolute memory position 277
function prototypes. 178
enforcing 223
function return addresses 128
function type information, omitting in object output. 219
FUNCTION (assembler directive) 131
function (pragma directive). 325, 339
functional (STL header file) 293
functions. 53
calling in different code models. 131
C++ and special function types 62
declaring 124, 178
inlining. 145, 174, 178, 258
interrupt. 55, 59
intrinsic 117, 178
monitor 59
omitting type info 219
parameters 126
placing in memory 167, 169
recursive
avoiding 178
storing data on stack 51
reentrancy (DLIB) 290
related extensions. 53
return values from 128
special function types. 55
trap. 57
verifying linked result 79
function_effects (pragma directive). 325, 339

G

getenv (library function), configuring support for 108
getw, in stdio.h 295
getzone (library function), configuring support for 109
getzone.c 109
__get_interrupt_state (intrinsic function) 275
__get_processor_register (intrinsic function) 56, 276
global arrays, accessing 135
global pointer (GP) 46
global variables
  accessing 135
  initialization 73
GLOBAL_AC (segment) 307
GLOBAL_AN (segment) 307
--guard_calls (compiler option) 208
guidelines, reading 25

H

__halt (intrinsic function) 277
HALT (assembler instruction) 277
Harbison, Samuel P. 28
hardware conflict, in V850Ex/xxx devices 204
hardware support in compiler 81
hash_map (STL header file) 293
hash_set (STL header file) 293
__has_constructor, symbol used in library 297
__has_destructor, symbol used in library 297
hdrstop (pragma directive) 325, 339
header files
  C 291
  C++ 292–293
  library 289
  special function registers 181
  STL 293
  DLib_Defaults.h 95, 100
  including stdbool.h for bool 228
  including stddef.h for wchar_t 229
header names, implementation-defined behavior 323
--header_context (compiler option) 209
heap
  dynamic memory 52
  segments for 75
  storing data 44
  VLA allocated on 226
heap segments
  HEAP (segment) 308
  placing 76
heap size
  and standard I/O 76
  changing default 76
HEAP (segment) 75, 308
heap (zero-sized), implementation-defined behavior 329
hints
  for good code generation 177
  implementation-defined behavior 322
  using efficient data types 163
__huge (extended keyword) 245
huge memory, accessing using assembler 138
huge (memory type) 47
HUGE_C (segment) 308
HUGE_I (segment) 308
HUGE_ID (segment) 309
HUGE_N (segment) 309
HUGE_Z (segment) 309

I

-I (compiler option) 209
IAR Command Line Build Utility 94
IAR Systems Technical Support 190
iarbuild.exe (utility) 94
__iar_cos_accurate (library routine) 111
__iar_cos_accuratef (library routine) 112
__iar_cos_accuratef (library function) 290
__iar_cos_accuratel (library routine) 112
__iar_cos_accuratel (library function) 290

__iar_cos_small (library routine) 110
__iar_cos_smallf (library routine). 111
__iar_cos_smalll (library routine). 111
__iar_exp_small (library routine) 110
__iar_exp_smallf (library routine) 111
__iar_exp_smalll (library routine) 111
__iar_log_small (library routine) 110
__iar_log_smallf (library routine). 111
__iar_log_smalll (library routine). 111
__iar_log10_small (library routine) 110
__iar_log10_smallf (library routine). 111
__iar_log10_smalll (library routine). 111
__iar_Powf (library routine) 112
__iar_Powl (library routine) 112
__iar_Pow_accurate (library routine) 111
__iar_pow_accurate (library routine) 111
__iar_Pow_accuratef (library routine) 112
__iar_pow_accuratef (library routine). 112
__iar_pow_accuratef (library function). 290
__iar_Pow_accuratel (library routine). 112
__iar_pow_accuratel (library routine). 112
__iar_pow_accuratel (library function). 290
__iar_pow_small (library routine). 110
__iar_pow_smallf (library routine) 111
__iar_pow_smalll (library routine) 111
__iar_program_start (label). 96
__iar_Sin (library routine) 110
__iar_Sinf (library routine) 112
__iar_Sinl (library routine) 112
__iar_Sin_accurate (library routine) 111
__iar_sin_accurate (library routine) 111
__iar_Sin_accuratef (library routine) 112
__iar_sin_accuratef (library routine). 112
__iar_sin_accuratef (library function). 290
__iar_Sin_accuratel (library routine) 112
__iar_sin_accuratel (library routine). 112
__iar_sin_accuratel (library function). 290
__iar_Sin_small (library routine) 110
__iar_sin_small (library routine). 110
__iar_Sin_smallf (library routine). 111
__iar_sin_smallf (library routine) 111
__iar_Sin_smalll (library routine). 111
__iar_sin_smalll (library routine) 111
__IAR_SYSTEMS_ICC__ (predefined symbol) 284
__iar_tan_accurate (library routine) 111
__iar_tan_accuratef (library routine). 112
__iar_tan_accuratef (library function). 290
__iar_tan_accuratel (library routine). 112
__iar_tan_accuratel (library function). 290
__iar_tan_small (library routine) 110
__iar_tan_smallf (library routine). 111
__iar_tan_smalll (library routine) 111
__ICCV850__ (predefined symbol) 284
ICODE (segment) 78, 310
icons, in this guide 30
IDE
  building a library from 95
  building applications from, an overview 36
identifiers, implementation-defined behavior 319
identifiers, implementation-defined behavior in C89 334
IEEE format, floating-point values 231
implementation-defined behavior
  C89 333
  Standard C 317
important_typedef (pragma directive). 325, 339
include files
  including before source files 221
  specifying 186
include_alias (pragma directive) 257
infinity 232
infinity (style for printing), implementation-defined behavior 328
inheritance, in Embedded C++ 153
initialization
  dynamic 95
  single-value 151
initialized data segments 73
initializers, static 150

inline assembler 119
avoiding 178
See also assembler language interface
inline functions 145
in compiler 174
inline (pragma directive) 258
inlining functions, implementation-defined behavior 322
installation directory 29
instantiate (pragma directive) 325, 339
instruction scheduling (compiler option) 176
int (data type) signed and unsigned 228
integer types 228
casting 233
implementation-defined behavior 320
intptr_t 233
ptrdiff_t 233
size_t 233
uintptr_t 233
integers, implementation-defined behavior in C89 335
integral promotion 179
internal error 190
__interrupt (extended keyword) 56, 245
using in pragma directives 269
interrupt functions 55
placement in memory 77
interrupt handler. *See* interrupt service routine
interrupt service routine 55
interrupt state, restoring 279
interrupt vector 55
specifying with pragma directive 269
interrupt vector table 55, 57
in linker configuration file 77
INTVEC segment 310
interrupts
disabling 246
during function execution 59
processor state 51
using with EC++ destructors 157
intptr_t (integer type) 233
__intrinsic (extended keyword) 246
intrinsic functions 178
fpu_sqrt_double 274
fpu_sqrt_float 275
overview 117
summary 271
__absolute_to_pic 272
example 65
__code_distance 273
example 65
__compare_and_exchange_for_interlock 273
__get_processor_register 276
example 56
__pic_to_absolute 277
example 65
__saturated_add 277
__saturated_sub 278
__search_ones_left 278
__search_ones_right 278
__search_zeros_left 279
__search_zeros_right 279
__set_processor_register 279
example 57
__synchronize_exceptions 280
__synchronize_memory 280
__synchronize_pipeline 280
__upper_mul64 280
intrinsics.h (header file) 271
inttypes.h (library header file) 292
INTVEC (segment) 77, 310
invocation syntax 185
iomanip (library header file) 293
ios (library header file) 293
iosfwd (library header file) 293
iostream (library header file) 293
iso646.h (library header file) 292
istream (library header file) 293
italic style, in this guide 30
iterator (STL header file) 293

I/O register. *See* SFR

J

Josuttis, Nicolai M. 28

K

keep_definition (pragma directive) 325, 339
Kernighan & Ritchie function declarations 179
    disallowing. 223
Kernighan, Brian W. 28
keywords. 239
    extended, overview of 41

L

-l (compiler option). 209
    for creating skeleton code 122
labels. 151
    assembler, making public. 222
    __iar_program_start. 96
    __program_start. 96
labels, in assembler. 132
Labrosse, Jean J. 28
Lajoie, Josée 28
language extensions
    Embedded C++ 153
    enabling using pragma 258
    enabling (-e). 206
language overview 35
language (pragma directive) 258
large code model. 54
    function calls 133
large with saddr (data model) 201
large (code model) 54
large (data model). 201
LD (assembler instruction) 134, 138
libraries
    definition of 36
    standard template library 293
    using a prebuilt 83
library configuration files
    DLIB 100
    DLib_Defaults.h 95, 100
    modifying 95
    specifying 205
library documentation 289
library features, missing from Embedded C++ 154
library functions 289
    summary, DLIB 291
library header files 289
library modules
    creating 210
    overriding. 93
library object files 289
library options, setting 40
library project template. 39
    using 94
library_default_requirements (pragma directive) . . . 325, 339
--library_module (compiler option) 210
library_provides (pragma directive) 325, 339
library_requirement_override (pragma directive) . . . 325, 339
lightbulb icon, in this guide. 30
limits.h (library header file) 292
__LINE__ (predefined symbol) 285
link register, considerations. 126
linkage, C and C++. 124
linker configuration file 68
    customizing 68
    using the -P command 70
    using the -Z command 70
linker map file. 79
linker output files 37
linker segment. *See* segment
linking
    from the command line 37

required input . . . 37
Lippman, Stanley B. . . . 28
list (STL header file) . . . 293
listing, generating . . . 209
literals, compound . . . 145
literature, recommended . . . 28
__LITTLE_ENDIAN__ (predefined symbol) . . . 285
local variables, *See* auto variables
locale
adding support for in library . . . 107
changing at runtime . . . 107
implementation-defined behavior . . . 320, 330
removing support for . . . 107
support for . . . 106
locale.h (library header file) . . . 292
located data segments . . . 76
located data, declaring extern . . . 169
location (pragma directive) . . . 168, 259
LOCFRAME (assembler directive) . . . 131
locking, of registers . . . 176
--lock_regs (compiler option) . . . 176, 211
--lock_regs_compatibility (compiler option) . . . 211
log (library routine) . . . 110
logf (library routine) . . . 111
logl (library routine) . . . 111
log10 (library routine) . . . 110
log10f (library routine) . . . 111
log10l (library routine) . . . 111
long double (data type) . . . 231
long float (data type), synonym for double . . . 150
long long (data type)
restrictions . . . 228
long long (data type) signed and unsigned . . . 228
long (data type) signed and unsigned . . . 228
longjmp, restrictions for using . . . 291
loop unrolling (compiler transformation) . . . 173
disabling . . . 218
loop-invariant expressions . . . 174
__low_level_init . . . 96
customizing . . . 99
low_level_init.c . . . 96
low-level processor operations . . . 146, 271
accessing . . . 117
__lseek (library function) . . . 105

M

macros
embedded in #pragma optimize . . . 262
ERANGE (in errno.h) . . . 326, 340
inclusion of assert . . . 286
NULL, implementation-defined behavior . . . 327
in C89 for DLIB . . . 340
substituted in #pragma directives . . . 146
variadic . . . 145
--macro_positions_in_diagnostics (compiler option) . . . 212
main (function)
definition (C89) . . . 333
implementation-defined behavior . . . 318
malloc (library function)
See also heap . . . 52
implementation-defined behavior in C89 . . . 342
Mann, Bernhard . . . 28
map (STL header file) . . . 293
map, linker . . . 79
math functions rounding mode,
implementation-defined behavior . . . 330
math functions (library functions) . . . 110
math.h (library header file) . . . 292
MB_LEN_MAX, implementation-defined behavior . . . 330
medium with saddr (data model) . . . 201
medium (data model) . . . 201
memory
access methods using assembler
base relative (brel) . . . 137
base-relative23 . . . 138
huge . . . 138

near 136
short address (saddr) 139
accessing 38, 45, 134
allocating in C++ 52
dynamic 52
heap 52
no bit accessing 139
non-initialized 181
RAM, saving 178
releasing in C++. 52
stack. 50
saving 178
used by global or static variables 43
memory layout, V850 43
memory management, type-safe 153
memory map
customizing the linker configuration file for 69
memory placement
using pragma directive 49
using type definitions. 49, 241
memory segment. *See* segment
memory types 45
brel (base-relative) 46
brel23. 47
C++ 50
huge 47
near 46
placing variables in 50
saddr (short address) 47
specifying 47
structures 49
summary 48
memory (pragma directive). 325, 339
memory (STL header file). 293
__memory_of
symbol used in library 297
message (pragma directive). 260
messages
disabling 224
forcing 260
Meyers, Scott 28
--mfc (compiler option). 212
--migration_preprocessor_extensions (compiler option). . 212
migration, from earlier IAR compilers 28
MISRA C, documentation 28
--misrac_verbose (compiler option) 195
--misrac1998 (compiler option) 195
--misrac2004 (compiler option) 195
mode changing, implementation-defined behavior 328
module consistency. 112
rtmodel. 264
module map, in linker map file 79
module name, specifying (--module_name) 213
module summary, in linker map file 79
--module_name (compiler option) 213
module_name (pragma directive) 325, 339
__monitor (extended keyword) 246
monitor functions 59, 246
MOVHI (assembler instruction) 138
multibyte character support. 207
multibyte characters, implementation-defined behavior 319, 331
multiple inheritance
in Extended EC++ 154
missing from Embedded C++ 153
missing from STL 154
multi-file compilation 171
mutable attribute, in Extended EC++ 154, 158

N

names block (call frame information). 140
namespace support
in Extended EC++ 154, 158
missing from Embedded C++ 154
naming conventions 30
NaN
implementation of 232

implementation-defined behavior. 328
native environment,
implementation-defined behavior 331
NDEBUG (preprocessor symbol) 286
__near (extended keyword). 246
near memory 136
near (memory type). 46
NEAR_C (segment) 310
NEAR_I (segment). 310
NEAR_ID (segment) 311
NEAR_N (segment) 311
NEAR_Z (segment) 311
new (keyword) 52
new (library header file) 293
no bit access (memory restriction) 139
non-initialized variables, hints for. 182
non-scalar parameters, avoiding 178
NOP (assembler instruction). 277
__noreturn (extended keyword) 247
Normal code model. 54
Normal DLIB (library configuration) 100
normal (code model). 54
Not a number (NaN) 232
NOT1 (assembler instruction). 134
__no_bit_access (extended keyword) 247
--no_clustering (compiler option) 213
--no_code_motion (compiler option) 214
--no_cross_call (compiler option). 214
--no_cse (compiler option) 214
--no_data_model_attribute (compiler option) 215
no_epilogue (pragma directive). 260
__no_init (extended keyword) 182, 247
--no_inline (compiler option) 215
__no_operation (intrinsic function). 277
--no_path_in_file_macros (compiler option). 215
no_pch (pragma directive) 325, 339
--no_scheduling (compiler option) 216
--no_size_constraints (compiler option) 216
--no_static_destruction (compiler option). 216
--no_system_include (compiler option) 217
--no_tbaa (compiler option) 217
--no_typedefs_in_diagnostics (compiler option). 217
--no_unroll (compiler option) 218
--no_warnings (compiler option) 218
--no_wrap_diagnostics (compiler option) 218
NULL
implementation-defined behavior. 327
implementation-defined behavior in C89 (DLIB) 340
pointer constant, relaxation to Standard C 150
numeric conversion functions,
implementation-defined behavior 331
numeric (STL header file). 293

O

-O (compiler option). 219
-o (compiler option) 220
object attributes. 241
object filename, specifying (-o) 220
object module name, specifying (--module_name) 213
object_attribute (pragma directive) 182, 261
--omit_types (compiler option) 219
once (pragma directive) 325, 339
--only_stdout (compiler option) 220
__open (library function) 105
operators
See also @ (operator)
for cast
in Extended EC++. 154
missing from Embedded C++ 154
for segment control 149
in inline assembler 119
precision for 32-bit float 231
precision for 64-bit float 232
sizeof, implementation-defined behavior 330
variants for cast 158
_Pragma (preprocessor) 145
__ALIGNOF__, for alignment control. 148
?, language extensions for 159

optimization
- clustering, disabling . . . 213
- code motion, disabling . . . 214
- common sub-expression elimination, disabling . . . 214
- configuration . . . 39
- disabling . . . 173
- function inlining, disabling (--no_inline) . . . 215
- hints . . . 177
- loop unrolling, disabling . . . 218
- scheduling, disabling . . . 216
- specifying (-O) . . . 219
- techniques . . . 173
- type-based alias analysis, disabling (--tbaa) . . . 217
- using inline assembler code . . . 120
- using pragma directive . . . 261

optimization levels . . . 172
optimize (pragma directive) . . . 261
option parameters . . . 191
options, compiler. *See* compiler options
Oram, Andy . . . 28
ostream (library header file) . . . 293
output
- from preprocessor . . . 221
- specifying for linker . . . 37

--output (compiler option) . . . 220
overhead, reducing . . . 173–174

P

pack (pragma directive) . . . 234, 262
packed structure types . . . 234
parameters
- function . . . 126
- hidden . . . 126
- non-scalar, avoiding . . . 178
- register . . . 126–127
- rules for specifying a file or directory . . . 192
- specifying . . . 193
- stack . . . 126–127
- typographic convention . . . 30

part number, of this guide . . . 2
permanent registers . . . 125
perror (library function), implementation-defined behavior in C89 . . . 342
__pic_to_absolute (intrinsic function) . . . 65
placement
- code and data . . . 299
- in named segments . . . 169

plain char, implementation-defined behavior . . . 319
pointer types . . . 232
- mixing . . . 150

pointers
- casting . . . 233
- data . . . 232
- function . . . 232
- global . . . 46
- implementation-defined behavior . . . 322
- implementation-defined behavior in C89 . . . 336

polymorphism, in Embedded C++ . . . 153
porting, code containing pragma directives . . . 252
position-independent code . . . 64
position-independent code model . . . 54
- function calls . . . 133

pow (library routine) . . . 110–111
- alternative implementation of . . . 290

powf (library routine) . . . 111–112
powl (library routine) . . . 111–112
pragma directives . . . 41
- summary . . . 251
- for absolute located data . . . 168
- list of all recognized . . . 324
- list of all recognized (C89) . . . 339
- pack . . . 234, 262
- type_attribute, using . . . 49
- vector . . . 58–59

_Pragma (preprocessor operator) . . . 145
predefined symbols
- overview . . . 41
- summary . . . 282

--predef_macro (compiler option) . . . 220
--preinclude (compiler option) . . . 221
--preprocess (compiler option) . . . 221
preprocessor
operator (_Pragma) . . . 145
output . . . 221
overview of . . . 281
preprocessor directives
comments at the end of . . . 151
implementation-defined behavior . . . 323
implementation-defined behavior in C89 . . . 338
#pragma . . . 251
preprocessor extensions
compatibility . . . 212
__VA_ARGS__ . . . 145
#warning message . . . 287
preprocessor symbols . . . 282
defining . . . 200
preserved registers . . . 125
__PRETTY_FUNCTION__ (predefined symbol) . . . 285
primitives, for special functions . . . 55
print formatter, selecting . . . 88
printf (library function) . . . 87
choosing formatter . . . 87
configuration symbols . . . 103
implementation-defined behavior . . . 328
implementation-defined behavior in C89 . . . 342
__printf_args (pragma directive) . . . 263
printing characters, implementation-defined behavior . . . 331
processor configuration . . . 38
processor operations
accessing . . . 117
low-level . . . 146, 271
processor, writing to (__set_processor_register) . . . 279
program entry label . . . 96
program termination, implementation-defined behavior . . . 318
programming hints . . . 177
__program_start (label) . . . 96
program, *see also* application
projects
basic settings for . . . 37
setting up for a library . . . 94
prototypes, enforcing . . . 223
ptrdiff_t (integer type) . . . 233
PUBLIC (assembler directive) . . . 222
publication date, of this guide . . . 2
--public_equ (compiler option) . . . 221
public_equ (pragma directive) . . . 325, 339
putenv (library function), absent from DLIB . . . 108
putw, in stdio.h . . . 295

Q

qualifiers
const and volatile . . . 235
implementation-defined behavior . . . 323
implementation-defined behavior in C89 . . . 337
queue (STL header file) . . . 294

R

-r (compiler option) . . . 201
raise (library function), configuring support for . . . 109
raise.c . . . 109
RAM
non-zero initialized variables . . . 73
saving memory . . . 178
range errors, in linker . . . 79
RCODE (segment) . . . 312
__read (library function) . . . 105
customizing . . . 101
read formatter, selecting . . . 89
reading guidelines . . . 25
reading, recommended . . . 28
realloc (library function) . . . 52
implementation-defined behavior in C89 . . . 342
See also heap

recursive functions
  avoiding 178
  storing data on stack 51
reentrancy (DLIB). 290
reference information, typographic convention 30
register constants 176
register keyword, implementation-defined behavior 322
register parameters 126–127
registered trademarks 2
registers
  assigning to parameters 127
  brel base
    R4 . . . . . 46, 126, 137
    R25 . . . . . 46, 126, 137
  callee-save, stored on stack 51
  implementation-defined behavior in C89 337
  in assembler-level routines 124
  loading permanently 176
  locking 176
  preserved 125
  processor, writing to (__set_processor_register) 279
  scratch 125
--reg_const (compiler option) 222
__reg_ep (runtime model attribute). 114
__reg_lock2 (runtime model attribute) 115
__reg_lock6 (runtime model attribute) 115
__reg_lock10 (runtime model attribute) 115
__reg_r25 (runtime model attribute) 115
reinterpret_cast (cast operator) 154
--relaxed_fp (compiler option) 222
remark (diagnostic message). 189
  classifying for compiler 203
  enabling in compiler 223
--remarks (compiler option) 223
remove (library function) 105
  implementation-defined behavior. 328
  implementation-defined behavior in C89 (DLIB) 341
remquo, magnitude of 326
rename (library function) 105
  implementation-defined behavior. 328
  implementation-defined behavior in C89 (DLIB) 341
__ReportAssert (library function) 112
required (pragma directive). 264
--require_prototypes (compiler option) 223
return addresses 128
return values, from functions 128
Ritchie, Dennis M. 28
__root (extended keyword) 247
routines, time-critical 117, 146, 271
RTE (assembler instruction) 249
rtmodel (assembler directive) 113
rtmodel (pragma directive) 264
rtti support, missing from STL 154
__rt_version (runtime model attribute) 115
runtime environment
  DLIB 81
  setting options for 40
  setting up (DLIB). 82
runtime libraries (DLIB)
  introduction 289
  customizing system startup code 99
  customizing without rebuilding 86
  filename syntax 86
  overriding modules in 93
  using prebuilt 83
runtime library
  setting up from command line 40
  setting up from IDE 40
runtime model attributes 112
  __cpu 114
  __data_model. 114
  __fpu_double 114
  __fpu_single 114
  __reg_ep 114
  __reg_lock2 115
  __reg_lock6 115
  __reg_lock10 115

__reg_r25 . . . 115
runtime model definitions . . . 265
runtime type information, missing from Embedded C++ . 153
R4 (brel base register) . . . 46, 137
considerations . . . 126
R18, constant value loaded into . . . 176
R19, constant value loaded into . . . 176
R25 (brel base register) . . . 46, 137
considerations . . . 126

S

__saddr (extended keyword) . . . 248
saddr memory . . . 47
accessing using assembler . . . 139
saddr (memory type) . . . 47
__SADDR_ACTIVE__ (predefined symbol) . . . 285
SADDR_BASE (segment) . . . 312
SADDR7_I (segment) . . . 312
SADDR7_ID (segment) . . . 313
SADDR7_N (segment) . . . 313
SADDR7_Z (segment) . . . 313
SADDR8_I (segment) . . . 314
SADDR8_ID (segment) . . . 314
SADDR8_N (segment) . . . 314
SADDR8_Z (segment) . . . 315
scanf (library function)
choosing formatter (DLIB) . . . 88
configuration symbols . . . 103
implementation-defined behavior . . . 329
implementation-defined behavior in C89 (DLIB) . . . 342
__scanf_args (pragma directive) . . . 265
scheduling (compiler transformation) . . . 176
disabling . . . 216
SCH0L (assembler instruction) . . . 279
SCH0R (assembler instruction) . . . 279
SCH1L (assembler instruction) . . . 278
SCH1R (assembler instruction) . . . 278
scratch registers . . . 125
__search_ones_left (intrinsic function) . . . 278
__search_ones_right (intrinsic function) . . . 278
__search_zeros_left (intrinsic function) . . . 279
__search_zeros_right (intrinsic function) . . . 279
section (pragma directive) . . . 265
segment group name . . . 71
segment map, in linker map file . . . 79
segment memory types, in XLINK . . . 68
segment (pragma directive) . . . 265
segments . . . 299
CLTCODE . . . 78
CLTVEC . . . 78
CODE . . . 77
code . . . 77
data . . . 71
declaring (#pragma segment) . . . 266
definition of . . . 67
ICODE . . . 78
initialized data . . . 73
introduction . . . 67
located data . . . 76
naming . . . 72
packing in memory . . . 70
placing in sequence . . . 70
static memory . . . 71
summary . . . 299
SYSCALLCODE . . . 78
SYSCALLVEC . . . 78
too long for address range . . . 79
too long, in linker . . . 79
TRAPVEC . . . 78
__segment_begin (extended operator) . . . 149
__segment_end (extended operator) . . . 149
__segment_size (extended operator) . . . 149
semaphores
C example . . . 59
C++ example . . . 61
operations on . . . 246
set (STL header file) . . . 294

setjmp.h (library header file) 292
setlocale (library function) 107
settings, basic for project configuration 37
__set_interrupt_state (intrinsic function) 279
__set_processor_register (intrinsic function) 57, 279
SET1 (assembler instruction) 134
severity level, of diagnostic messages 189
  specifying 190
SFR
  accessing special function registers 181
  declaring extern special function registers 169
shared object 188
short address memory, accessing using assembler 139
short addressing
  testing if in use (__SADDR_ACTIVE__) 285
short (data type) 228
signal (library function)
  configuring support for 109
  implementation-defined behavior 326
  implementation-defined behavior in C89 340
signals, implementation-defined behavior 318
  at system startup 318
signal.c 109
signal.h (library header file) 292
signed char (data type) 228–229
  specifying 198
signed int (data type) 228
signed long long (data type) 228
signed long (data type) 228
signed short (data type) 228
--silent (compiler option) 224
silent operation
  specifying in compiler 224
sin (library function) 290
sin (library routine) 110–111
sinf (library routine) 111–112
sinl (library routine) 111–112
64-bits (floating-point format) 231
sizeof, using in preprocessor extensions 212
size_t (integer type) 233
skeleton code, creating for assembler language interface . 121
skeleton.s85 (assembler source output) 122
SLD instruction, causing hardware problems 204
SLD (assembler instruction) 134
slist (STL header file) 294
small with saddr (data model) 201
small (data model) 201
source files, list all referred 209
space characters, implementation-defined behavior 327
special function registers (SFR) 181
special function types 55
  overview 41
sprintf (library function) 87
  choosing formatter 87
SQRTF (FPU instruction) 274–275
sscanf (library function)
  choosing formatter (DLIB) 88
SST (assembler instruction) 134
sstream (library header file) 293
ST (assembler instruction) 134, 138
stack 50, 74
  advantages and problems using 51
  changing default size of 74
  cleaning after function return 128
  contents of 51
  layout 127
  saving space 178
  size 75
stack parameters 126–127
stack pointer 51
stack pointer register, considerations 126
stack segment
  CSTACK 306
  placing in memory 75
stack (STL header file) 294
Standard C
  implementation-defined behavior 317
  library compliance with 289

specifying strict usage 224
standard error
redirecting in compiler 220
standard input 101
standard output 101
specifying in compiler 220
standard template library (STL)
in C++ 293
in Extended EC++ 154, 158
missing from Embedded C++ 154
startup code
placement of 77
See also CSTART
startup system. *See* system startup
statements, implementation-defined behavior in C89 338
static clustering (compiler transformation) 175
static data, in configuration file 74
static memory segments 71
static overlay 131
static variables 43
initialization 73
taking the address of 178
static_assert() 148
static_cast (cast operator) 154
status flags for floating-point 295
std namespace, missing from EC++
and Extended EC++ 158
stdarg.h (library header file) 292
stdbool.h (library header file) 228, 292
__STDC__ (predefined symbol) 285
STDC CX_LIMITED_RANGE (pragma directive) 266
STDC FENV_ACCESS (pragma directive) 267
STDC FP_CONTRACT (pragma directive) 267
__STDC_VERSION__ (predefined symbol) 285
stddef.h (library header file) 229, 292
stderr 105, 220
stdin 105
implementation-defined behavior in C89 (DLIB) 341
stdint.h (library header file) 292, 294
stdio.h (library header file) 292
stdio.h, additional C functionality 295
stdlib.h (library header file) 292
stdout 105, 220
implementation-defined behavior 327
implementation-defined behavior in C89 (DLIB) 341
Steele, Guy L. 28
STL 158
strcasecmp, in string.h 296
strdup, in string.h 296
streambuf (library header file) 293
streams
implementation-defined behavior 318
supported in Embedded C++ 154
strerror (library function), implementation-defined
behavior 332
strerror (library function),
implementation-defined behavior in C89 (DLIB) 343
--strict (compiler option) 224
string (library header file) 293
strings, supported in Embedded C++ 154
string.h (library header file) 292
string.h, additional C functionality 296
strncasecmp, in string.h 296
strnlen, in string.h 296
Stroustrup, Bjarne 29
strstream (library header file) 293
strtod (library function), configuring support for 110
structure types
alignment 233–234
layout of 234
packed 234
structures
accessing using a pointer 135
aligning 263
anonymous 148, 165
implementation-defined behavior 323
implementation-defined behavior in C89 337
packing and unpacking 165
placing in memory type 49
subnormal numbers 231–232

support, technical 190
Sutter, Herb. 29
symbol names, using in preprocessor extensions 212
symbols
anonymous, creating 145
including in output. 264
listing in linker map file 79
overview of predefined. 41
preprocessor, defining 200
SYNCE (assembler instruction) 280
__synchronize_exceptions (intrinsic function) 280
__synchronize_memory (intrinsic function) 280
__synchronize_pipeline (intrinsic function) 280
SYNCM (assembler instruction). 280
SYNCP (assembler instruction) 280
syntax
command line options 191
extended keywords. 48, 240–242
invoking compiler 185
__syscall (extended keyword). 248
syscall functions
in C language. 58
placement in memory. 78
syscall table 248
syscall vector table, SYSCALLVEC segment. 315
syscall (function type). 58
SYSCALLCODE (segment). 78, 315
SYSCALLVEC (segment) 78, 315
system function, implementation-defined behavior. . 319, 329
system startup
customizing 99
DLIB 96
system termination
C-SPY interface to. 99
DLIB 98
system (library function)
configuring support for 108
implementation-defined behavior in C89 (DLIB) 343
system_include (pragma directive) 325, 339
--system_include_dir (compiler option) 224

T

tan (library function). 290
tan (library routine). 110–111
tanf (library routine) 111–112
tanl (library routine) 111–112
__task (extended keyword) 249
technical support, IAR Systems 190
template support
in Extended EC++ 154, 157
missing from Embedded C++ 153
Terminal I/O window
making available (DLIB) 91
not supported when 94
terminal I/O, debugger runtime interface for. 90
terminal output, speeding up 91
termination of system. *See* system termination
termination status, implementation-defined behavior 329
terminology. 29
tgmath.h (library header file) 292
32-bits (floating-point format) 231
this (pointer) 123
__TIME__ (predefined symbol) 286
time zone (library function)
implementation-defined behavior in C89 343
time zone (library function), implementation-defined
behavior 329
time-critical routines. 117, 146, 271
time.c 109
time.h (library header file) 292
additional C functionality. 296
time32 (library function), configuring support for 109
time64 (library function), configuring support for 109
tiny with saddr (data model) 201
tiny (data model). 200
tips, programming. 177
tools icon, in this guide 30

trademarks 2
transformations, compiler 170
translation, implementation-defined behavior 317
translation, implementation-defined behavior in C89 333
__trap (extended keyword) 57, 249
trap functions 57
trap vector table, TRAPVEC segment. 316
trap vectors
  placement in memory. 78
trap vectors, specifying with pragma directive 269
TRAP (assembler instruction). 249
TRAPVEC (segment) 78, 316
TST1 (assembler instruction) 134
type attributes 239
  specifying 268
type definitions, used for specifying memory storage . 49, 241
type information, omitting 219
type qualifiers
  const and volatile 235
  implementation-defined behavior. 323
  implementation-defined behavior in C89 337
typedefs
  excluding from diagnostics 217
  repeated 150
  using in preprocessor extensions 212
type_attribute (pragma directive) 49, 268
type-based alias analysis (compiler transformation) 174
  disabling 217
type-safe memory management 153
typographic conventions 30

U

UBROF
  format of linkable object files 187
  specifying, example of 37
uchar.h (library header file). 292
uintptr_t (integer type) 233
underflow errors, implementation-defined behavior 326
underflow range errors,
implementation-defined behavior in C89 340
__ungetchar, in stdio.h 295
unions
  anonymous. 148, 165
  implementation-defined behavior. 323
  implementation-defined behavior in C89 337
universal character names, implementation-defined
behavior 324
unroll (pragma directive) 268
unsigned char (data type) 228–229
  changing to signed char 198
unsigned int (data type). 228
unsigned long long (data type) 228
unsigned long (data type) 228
unsigned short (data type). 228
__upper_mul64 (intrinsic function) 280
--use_c++_inline (compiler option) 225
utility (STL header file) 294

V

variable declarations, C++ 46
variable type information, omitting in object output 219
variables
  auto 50–51
  defined inside a function 50
  global
    accessing. 135
    initialization of 73
    placement in memory 43
  hints for choosing 177
  local. *See* auto variables
  non-initialized 182
  omitting type info 219
  placing at absolute addresses 169
  placing in named segments 169
  static
    placement in memory 43

taking the address of . . . 178
static and global, initializing . . . 73
?CODE_DISTANCE . . . 97
variadic macros . . . 149
vector (pragma directive) . . . 56–59, 269
vector (STL header file) . . . 294
version
compiler subversion number . . . 286
of compiler . . . 286
version number
of this guide . . . 2
--vla (compiler option) . . . 226
void, pointers to . . . 150
volatile
and const, declaring objects . . . 237
declaring objects . . . 235
protecting simultaneously accesses variables . . . 180
rules for access . . . 236
V850
memory access . . . 38
memory layout . . . 43
supported devices . . . 36

W

#warning message (preprocessor extension) . . . 287
warnings . . . 189
classifying in compiler . . . 204
disabling in compiler . . . 218
exit code in compiler . . . 226
warnings icon, in this guide . . . 30
warnings (pragma directive) . . . 325, 340
--warnings_affect_exit_code (compiler option) . . . 188, 226
--warnings_are_errors (compiler option) . . . 226
wchar_t (data type), adding support for in C . . . 229
wchar.h (library header file) . . . 292, 295
wctype.h (library header file) . . . 292
web sites, recommended . . . 29
white-space characters, implementation-defined behavior 317
__write (library function) . . . 105
customizing . . . 101
__write_array, in stdio.h . . . 295
__write_buffered (DLIB library function) . . . 91

X

XLINK errors
range error . . . 79
segment too long . . . 79
XLINK segment memory types . . . 68
xreportassert.c . . . 112

Symbols

_Exit (library function) . . . 98
_exit (library function) . . . 98
__absolute_to_pic (intrinsic function) . . . 65, 272
__ALIGNOF__ (operator) . . . 148
__asm (language extension) . . . 119
__assignment_by_bitwise_copy_allowed, symbol used in library . . . 296
__BASE_FILE__ (predefined symbol) . . . 282
__brel (extended keyword) . . . 243
__brel23 (extended keyword) . . . 244
__BUILD_NUMBER__ (predefined symbol) . . . 282
__callt (extended keyword) . . . 58–59, 244
__close (library function) . . . 105
__code_distance (intrinsic function) . . . 65, 273
__code_model (runtime model attribute) . . . 114
__CODE_MODEL__ (predefined symbol) . . . 282
__code, symbol used in library . . . 296
__compare_and_exchange_for_interlock (intrinsic function) . . . 273
__constrange(), symbol used in library . . . 297
__construction_by_bitwise_copy_allowed, symbol used in library . . . 297
__CORE__ (predefined symbol) . . . 282

__cplusplus (predefined symbol) 282
__cpu (runtime model attribute) 114
__CPU__ (predefined symbol) 283
__data_model (runtime model attribute). 114
__DATA_MODEL__ (predefined symbol) 283
__DATE__ (predefined symbol) 283
__disable_interrupt (intrinsic function). 274
__DLIB_FILE_DESCRIPTOR (configuration symbol) . . 105
__embedded_cplusplus (predefined symbol) 283
__enable_interrupt (intrinsic function) 274
__exit (library function) 98
__FILE__ (predefined symbol). 283
__flat (extended keyword) 244
__fpu_double (runtime model attribute) 114
__fpu_single (runtime model attribute). 114
__fpu_sqrt_double (intrinsic function) 274
__fpu_sqrt_float (intrinsic function) 275
__FPU__ (predefined symbol) 284
__FUNCTION__ (predefined symbol) 152, 284
__func__ (predefined symbol) 152, 284
__gets, in stdio.h 295
__get_interrupt_state (intrinsic function) 275
__get_processor_register (intrinsic function) 56, 276
__halt (intrinsic function) 277
__has_constructor, symbol used in library 297
__has_destructor, symbol used in library 297
__huge (extended keyword) 245
__iar_cos_accurate (library routine) 111
__iar_cos_accuratef (library routine) 112
__iar_cos_accuratel (library routine) 112
__iar_cos_small (library routine) 110
__iar_cos_smallf (library routine). 111
__iar_cos_smalll (library routine). 111
__iar_exp_small (library routine) 110
__iar_exp_smallf (library routine) 111
__iar_exp_smalll (library routine) 111
__iar_log_small (library routine) 110
__iar_log_smallf (library routine). 111
__iar_log_smalll (library routine). 111
__iar_log10_small (library routine) 110
__iar_log10_smallf (library routine). 111
__iar_log10_smalll (library routine). 111
__iar_Pow (library routine). 111
__iar_Powf (library routine) 112
__iar_Powl (library routine) 112
__iar_Pow_accurate (library routine) 111
__iar_pow_accurate (library routine) 111
__iar_Pow_accuratef (library routine) 112
__iar_pow_accuratef (library routine). 112
__iar_Pow_accuratel (library routine). 112
__iar_pow_accuratel (library routine). 112
__iar_pow_small (library routine). 110
__iar_pow_smallf (library routine) 111
__iar_pow_smalll (library routine) 111
__iar_program_start (label). 96
__iar_Sin (library routine) 110–111
__iar_Sinf (library routine) 111–112
__iar_Sinl (library routine) 111–112
__iar_Sin_accurate (library routine) 111
__iar_sin_accurate (library routine) 111
__iar_Sin_accuratef (library routine) 112
__iar_sin_accuratef (library routine). 112
__iar_Sin_accuratel (library routine) 112
__iar_sin_accuratel (library routine). 112
__iar_Sin_small (library routine) 110
__iar_sin_small (library routine). 110
__iar_Sin_smallf (library routine). 111
__iar_sin_smallf (library routine) 111
__iar_Sin_smalll (library routine). 111
__iar_sin_smalll (library routine) 111
__IAR_SYSTEMS_ICC__ (predefined symbol) 284
__iar_tan_accurate (library routine) 111
__iar_tan_accuratef (library routine). 112
__iar_tan_accuratel (library routine). 112
__iar_tan_small (library routine) 110
__iar_tan_smallf (library routine) 111
__iar_tan_smalll (library routine) 111
__ICCV850__ (predefined symbol) 284

__interrupt (extended keyword) 56, 245
using in pragma directives 269
__intrinsic (extended keyword). 246
__LINE__ (predefined symbol) 285
__LITTLE_ENDIAN__ (predefined symbol). 285
__low_level_init 96
__low_level_init, customizing 99
__lseek (library function) 105
__memory_of
symbol used in library 297
__monitor (extended keyword) 246
__near (extended keyword). 246
__noreturn (extended keyword) 247
__no_bit_access (extended keyword) 247
__no_init (extended keyword) 182, 247
__no_operation (intrinsic function). 277
__open (library function) 105
__pic_to_absolute (intrinsic function). 65, 277
__PRETTY_FUNCTION__ (predefined symbol). 285
__printf_args (pragma directive). 263
__program_start (label). 96
__read (library function). 105
customizing 101
__reg_ep (runtime model attribute). 114
__reg_lock2 (runtime model attribute) 115
__reg_lock6 (runtime model attribute) 115
__reg_lock10 (runtime model attribute) 115
__reg_r25 (runtime model attribute). 115
__ReportAssert (library function). 112
__root (extended keyword) 247
__rt_version (runtime model attribute) 115
__saddr (extended keyword). 248
__SADDR_ACTIVE__ (predefined symbol) 285
__saturated_add (intrinsic function) 277
__saturated_sub (intrinsic function) 278
__scanf_args (pragma directive) 265
__search_ones_left (intrinsic function). 278
__search_ones_right (intrinsic function). 278
__search_zeros_left (intrinsic function) 279
__search_zeros_right (intrinsic function) 279
__segment_begin (extended operator 149
__segment_end (extended operators) 149
__segment_size (extended operators) 149
__set_interrupt_state (intrinsic function) 279
__set_processor_register (intrinsic function) 57, 279
__STDC_VERSION__ (predefined symbol) 285
__STDC__ (predefined symbol). 285
__synchronize_exceptions (intrinsic function) 280
__synchronize_memory (intrinsic function) 280
__synchronize_pipeline (intrinsic function) 280
__syscall (extended keyword). 248
__task (extended keyword) 249
__TIME__ (predefined symbol) 286
__trap (extended keyword) 57, 249
__ungetchar, in stdio.h 295
__upper_mul64 (intrinsic function) 280
__VA_ARGS__ (preprocessor extension). 145
__write (library function) 105
customizing 101
__write_array, in stdio.h 295
__write_buffered (DLIB library function). 91
-D (compiler option). 200
-e (compiler option) 206
-f (compiler option). 208
-I (compiler option). 209
-l (compiler option). 209
for creating skeleton code 122
-O (compiler option). 219
-o (compiler option) 220
-r (compiler option). 201
--aggressive_inlining (compiler option) 197
--aggressive_unrolling (compiler option) 197
--allow_misaligned_data_access (compiler option) 198
--char_is_signed (compiler option). 198
--char_is_unsigned (compiler option). 199
--code_model (compiler option) 199
--c89 (compiler option). 198
--data_model (compiler option) 200

--debug (compiler option) 201
--dependencies (compiler option) 201
--diagnostics_tables (compiler option) 204
--diag_error (compiler option) 202
--diag_remark (compiler option). 203
--diag_suppress (compiler option) 203
--diag_warning (compiler option) 204
--disable_sld_suppression (compiler option). 204
--discard_unused_publics (compiler option). 205
--dlib_config (compiler option). 205
--ec++ (compiler option). 206
--eec++ (compiler option). 206
--enable_multibytes (compiler option) 207
--error_limit (compiler option) 207
--fpu (compiler option) 208
--guard_calls (compiler option). 208
--header_context (compiler option). 209
--library_module (compiler option) 210
--lock_regs (compiler option) 176, 211
--lock_regs_compatibility (compiler option). 177, 211
--macro_positions_in_diagnostics (compiler option) 212
--mfc (compiler option). 212
--migration_preprocessor_extensions (compiler option) . . 212
--misrac_verbose (compiler option) 195
--misrac1998 (compiler option) 195
--misrac2004 (compiler option) 195
--module_name (compiler option) 213
--no_clustering (compiler option) 213
--no_code_motion (compiler option) 214
--no_cross_call (compiler option) 214
--no_cse (compiler option) 214
--no_data_model_attribute (compiler option) 215
--no_inline (compiler option) 215
--no_path_in_file_macros (compiler option). 215
--no_scheduling (compiler option) 216
--no_size_constraints (compiler option) 216
--no_static_destruction (compiler option). 216
--no_system_include (compiler option) 217
--no_typedefs_in_diagnostics (compiler option). 217
--no_unroll (compiler option) 218
--no_warnings (compiler option) 218
--no_wrap_diagnostics (compiler option) 218
--omit_types (compiler option) 219
--only_stdout (compiler option) 220
--output (compiler option). 220
--predef_macro (compiler option). 220
--preinclude (compiler option) 221
--preprocess (compiler option) 221
--reg_const (compiler option) 176, 222
--relaxed_fp (compiler option) 222
--remarks (compiler option) 223
--require_prototypes (compiler option). 223
--silent (compiler option) 224
--strict (compiler option). 224
--system_include_dir (compiler option) 224
--use_c++_inline (compiler option) 225
--vla (compiler option) 226
--warnings_affect_exit_code (compiler option) 188, 226
--warnings_are_errors (compiler option) 226
?BREL_BASE (assembler label) 46, 137
?BREL_CBASE (assembler label) 46, 137
?CODE_DISTANCE (assembler variable) 97, 134
?SADDR_BASE (assembler label). 139
?Springboard_R29 (run-time library routine) 132
@ (operator)
  placing at absolute address. 168
  placing in segments 169
#include files, specifying 186, 209
#warning message (preprocessor extension). 287
%Z replacement string,
implementation-defined behavior 330

Numerics

32-bits (floating-point format) 231
64-bits (floating-point format) 231