IAR C/C++ Development Guide

Compiling and Linking

for STMicroelectronics'
STM8 Microcontroller Family

DSTM8-2

COPYRIGHT NOTICE

Copyright © 2010 IAR Systems AB.

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

DISCLAIMER

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

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

TRADEMARKS

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

STMicroelectronics is a registered trademark of STMicroelectronics Corporation. STM8 is a trademark of STMicroelectronics Corporation.

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

EDITION NOTICE

Second edition: April 2010

Part number: DSTM8-2

This guide applies to version 1.x of IAR Embedded Workbench® for STM8.

Internal reference: M3, Too6.0, csrct2010.1, V_100308, IMAE.

Brief contents

Tables xxv

Preface xxvii

Part 1. Using the build tools 1

Introduction to the IAR build tools 3

Developing embedded applications 9

Data storage 23

Functions 35

Linking using ILINK 43

Linking your application 53

The DLIB runtime environment 65

Assembler language interface 95

Using C 117

Using C++ 127

Application-related considerations 139

Efficient coding for embedded applications 147

Part 2. Reference information 165

External interface details 167

Compiler options 177

Linker options 207

Data representation 225

Extended keywords 237

Pragma directives 249
Intrinsic functions 265
The preprocessor 269
Library functions 275
The linker configuration file 283
Section reference 305
IAR utilities 315
Implementation-defined behavior 345
Index 361

Contents

Tables ... xxv

Preface ... xxvii

Who should read this guide ... xxvii

How to use this guide ... xxvii

What this guide contains ... xxviii

Other documentation ... xxix

Further reading ... xxx

Document conventions ... xxx

Typographic conventions ... xxxi

Naming conventions ... xxxi

Part 1. Using the build tools ... 1

Introduction to the IAR build tools ... 3

The IAR build tools—an overview ... 3

IAR C/C++ Compiler ... 3

IAR Assembler ... 4

The IAR ILINK Linker ... 4

Specific ELF tools ... 4

External tools ... 4

IAR language overview ... 5

Device support ... 5

Supported STM8 devices ... 6

Preconfigured support files ... 6

Examples for getting started ... 6

Special support for embedded systems ... 6

Extended keywords ... 7

Pragma directives ... 7

Predefined symbols ... 7

Special function types ... 7

Accessing low-level features ... 7

Developing embedded applications 9

Developing embedded software using IAR build tools 9

Mapping of internal and external memory 9

Communication with peripheral units 10

Event handling 10

System startup 10

Real-time operating systems 10

The build process—an overview 11

The translation process 11

The linking process 12

After linking 13

Application execution—an overview 14

The initialization phase 14

The execution phase 18

The termination phase 18

Building applications—an overview 18

Basic project configuration 19

Processor configuration 19

Data model 19

Code model 20

Optimization for speed and size 20

Runtime environment 21

Data storage 23

Introduction 23

Different ways to store data 23

Data models 24

Specifying a data model 24

Memory types 26

Tiny 26

Near 26

Far 27

Huge 27

Eeprom 27

Using data memory attributes ... 27
Pointers and memory types ... 29
Structures and memory types ... 30
More examples ... 30
C++ and memory types ... 31
Auto variables—on the stack ... 31
The stack ... 31
Dynamic memory on the heap ... 33

Functions ... 35

Function-related extensions ... 35
Code models and memory attributes for function storage 35
Using function memory attributes ... 36
Primitives for interrupts, concurrency, and OS-related programming ... 37
Interrupt functions ... 37
Monitor functions ... 38
C++ and special function types ... 41

Linking using ILINK ... 43

Linking—an overview ... 43
Modules and sections ... 43
The linking process ... 44
Placing code and data—the linker configuration file ... 46
A simple example of a configuration file ... 47
Initialization at system startup ... 49
The initialization process ... 49
C++ dynamic initialization ... 50

Linking your application ... 53

Linking considerations ... 53
Choosing a linker configuration file ... 53
Defining your own memory areas ... 54
Placing sections ... 55
Reserving space in RAM ... 56

Keeping modules ... 57
Keeping symbols and sections ... 57
Application startup ... 57
Setting up the stack ... 58
Setting up the heap ... 58
Setting up the atexit limit ... 58
Changing the default initialization ... 58
Interaction between ILINK and the application ... 61
Standard library handling ... 61
Producing other output formats than ELF/DWARF ... 62
Hints for troubleshooting ... 62
Relocation errors ... 62
The DLIB runtime environment ... 65
Introduction to the runtime environment ... 65
Runtime environment functionality ... 65
Setting up the runtime environment ... 66
Using a prebuilt library ... 67
Choosing a library ... 67
Groups of library files ... 67
Customizing a prebuilt library without rebuilding ... 69
Choosing formatters for printf and scanf ... 69
Choosing printf formatter ... 69
Choosing scanf formatter ... 70
Application debug support ... 71
Including C-SPY debugginG support ... 71
The debug library functionality ... 72
The C-SPY Terminal I/O window ... 73
Overriding library modules ... 74
Building and using a customized library ... 75
Setting up a library project ... 75
Modifying the library functionality ... 76
Using a customized library ... 76

System startup and termination 77
System startup 77
System termination 79
Customizing system initialization 80
__low_level_init 80
Modifying the file cstartup.s 81
Library configurations 81
Choosing a runtime configuration 81
Standard streams for input and output 82
Implementing low-level character input and output 82
Configuration symbols for printf and scanf 84
Customizing formatting capabilities 85
File input and output 85
Locale 86
Locale support in prebuilt libraries 86
Customizing the locale support 87
Changing locales at runtime 88
Environment interaction 88
The getenv function 89
The system function 89
Signal and raise 89
Time 90
Strtod 90
Pow 90
Assert 91
Atexit 91
Checking module consistency 91
Runtime model attributes 91
Using runtime model attributes 92
Predefined runtime attributes 93
User-defined runtime model attributes 93

Assembler language interface 95

Mixing C and assembler 95

Intrinsic functions 95

Mixing C and assembler modules 96

Inline assembler 97

Calling assembler routines from C 98

Creating skeleton code 98

Compiling the code 99

Calling assembler routines from C++ 100

Calling convention 101

Function declarations 102

Using C linkage in C++ source code 102

Virtual registers 102

Preserved versus scratch registers 103

Function entrance 104

Function exit 106

Restrictions for special function types 106

Examples 107

Calling functions 108

Assembler instructions used for calling functions 108

Memory access methods 109

The tiny memory access method 110

The near memory access method 111

The far memory access method 111

The huge memory access method 111

The eeprom memory access method 112

Call frame information 113

CFI directives 113

Creating assembler source with CFI support 114

Using C 117

C language overview 117

Extensions overview 118

Enabling language extensions 119

IAR C language extensions 120
Extensions for embedded systems programming 120
Relaxations to Standard C 122

Using C++ 127

Overview 127
Standard Embedded C++ 127
Extended Embedded C++ 128
Enabling C++ support 128

Feature descriptions 129
Classes 129
Function types 132
Templates 132
Variants of cast operators 134
Mutable 135
Namespace 135
The STD namespace 135
Pointer to member functions 135
Using interrupts and EC++ destructors 135

C++ language extensions 136

Application-related considerations 139

Output format considerations 139
Stack considerations 139
Stack size considerations 140
Heap considerations 140
Interaction between the tools and your application 140
Checksum calculation 142
Calculating a checksum 143
Adding a checksum function to your source code 144
Things to remember 145
C-SPY considerations 146

Efficient coding for embedded applications 147
Selecting data types 147
Using efficient data types 147
Floating-point types 148
Using the best pointer type 148
Anonymous structs and unions 148
Controlling data and function placement in memory 150
Data placement at an absolute location 151
Data and function placement in sections 152
Controlling compiler optimizations 153
Scope for performed optimizations 153
Optimization levels 154
Speed versus size 155
Fine-tuning enabled transformations 155
Facilitating good code generation 158
Writing optimization-friendly source code 158
Efficient use of memory types 159
Saving stack space and RAM memory 160
Function prototypes 160
Integer types and bit negation 161
Protecting simultaneously accessed variables 161
Accessing special function registers 162
Non-initialized variables 163
Part 2. Reference information 165
External interface details 167
Invocation syntax 167
Compiler invocation syntax 167
ILINK invocation syntax 167
Passing options 168
Environment variables 169

Include file search procedure 169
Compiler output 170
ILINK output 172
Diagnostics 172
Message format for the compiler 173
Message format for the linker 173
Severity levels 174
Setting the severity level 174
Internal error 174

Compiler options 177

Options syntax 177
Types of options 177
Rules for specifying parameters 177
Summary of compiler options 180
Descriptions of options 182
--c89 182
--char_is_signed 183
--char_is_unsigned 183
--code_model 183
--core 184
-D 184
--data_model 185
--debug, -r 185
--dependencies 185
--diag_error 186
--diag_remark 187
--diag_suppress 187
--diag_warning 188
--diagnostics_tables 188
--discard_unused_publics 188
--dlib_config 189
-e 190
--ec++ 190

--eec++ ... 190
--enable_multibytes ... 190
--error_limit ... 191
-f ... 191
--header_context ... 192
-I ... 192
-l ... 192
--mfc ... 193
--no_code_motion ... 194
--no_cross_call ... 194
--no_cse ... 194
--no_fragments ... 195
--no_inline ... 195
--no_path_in_file_macros ... 195
--no_static_destruction ... 196
--no_system_include ... 196
--no_tbaa ... 196
--no_typedefs_in_diagnostics ... 197
--no_unroll ... 197
--no_warnings ... 198
--no_wrap_diagnostics ... 198
-O ... 198
--only_stdout ... 199
--output, -o ... 199
--predef_macros ... 200
--preinclude ... 200
--preprocess ... 200
--public_equ ... 201
--relaxed_fp ... 201
--remarks ... 202
--require_prototypes ... 202
--silent ... 202
--strict ... 203
--system_include_dir ... 203

--use_unix_directory_separators ... 203
--vla ... 204
--vregs ... 204
--warnings_affect_exit_code ... 204
--warnings_are_errors ... 205

Linker options ... 207

Summary of linker options ... 207
Descriptions of options ... 209
--config ... 209
--config_def ... 209
--cpp_init_routine ... 210
--debug_lib ... 210
--define_symbol ... 210
--dependencies ... 211
--diag_error ... 211
--diag_remark ... 212
--diag_suppress ... 212
--diag_warning ... 213
--diagnostics_tables ... 213
--entry ... 213
--error_limit ... 214
--export_builtin_config ... 214
-f ... 214
--force_output ... 215
--image_input ... 215
--keep ... 216
--log ... 216
--log_file ... 216
--mangled_names_in_messages ... 217
--map ... 217
--no_fragments ... 218
--no_library_search ... 218
--no_locals ... 219

--no_range_reservations 219
--no_remove 219
--no_warnings 219
--no_wrap_diagnostics 220
--only_stdout 220
--output, -o 220
--place_holder 220
--redirect 221
--remarks 221
--silent 222
--strip 222
--warnings_affect_exit_code 222
--warnings_are_errors 222

Data representation 225

Alignment 225
Alignment on the STM8 microcontroller 225
Byte order 226
Basic data types 226
Integer types 226
Floating-point types 230
Pointer types 231
Function pointers 231
Data pointers 232
Casting 232
Structure types 233
General layout 233
Type qualifiers 234
Declaring objects volatile 234
Declaring objects volatile and const 235
Declaring objects const 235

Data types in C++ 236

Extended keywords 237

General syntax rules for extended keywords 237

Type attributes 237

Object attributes 240

Summary of extended keywords 241

Descriptions of extended keywords 242

__eeprom 242

__far 242

__far_func 243

__huge 243

__huge_func 243

__interrupt 244

__intrinsic 244

__monitor 244

__near 245

__near_func 245

__no_init 246

__noreturn 246

__root 246

__task 247

__tiny 247

__weak 247

Pragma directives 249

Summary of pragma directives 249

Descriptions of pragma directives 250

basic_template_matching 250

bitfields 251

data_alignment 252

diag_default 252

diag_error 252

diag_remark 253

diag_suppress 253

diag_warning 253
error 254
include_alias 254
inline 255
language 256
location 257
message 257
object_attribute 258
optimize 258
__printf_args 259
required 259
rtmodel 260
__scanf_args 261
section 261
STDC CX_LIMITED_RANGE 262
STDC FENV_ACCESS 262
STDC FP_CONTRACT 262
type_attribute 263
vector 263
weak 264

Intrinsic functions 265

Summary of intrinsic functions 265
Descriptions of intrinsic functions 266
__disable_interrupt 266
__enable_interrupt 266
__get_interrupt_state 266
__halt 267
__no_operation 267
__set_interrupt_state 267
__trap 267
__wait_for_exception 267
__wait_for_interrupt 267

The preprocessor 269

Overview of the preprocessor 269

Descriptions of predefined preprocessor symbols 270

Descriptions of miscellaneous preprocessor extensions 272

NDEBUG 272

#warning message 273

Library functions 275

Library overview 275

Header files 275

Library object files 275

Reentrancy 276

IAR DLIB Library 276

C header files 277

C++ header files 278

Library functions as intrinsic functions 280

Added C functionality 280

The linker configuration file 283

Overview 283

Defining memories and regions 284

Define memory directive 284

Define region directive 285

Regions 285

Region literal 285

Region expression 287

Empty region 288

Section handling 288

Define block directive 289

Define overlay directive 290

Initialize directive 291

Do not initialize directive 294

Keep directive 294

Place at directive 295

Place in directive ... 296
Section selection ... 296
Section-selectors ... 297
Extended-selectors ... 299
Using symbols, expressions, and numbers ... 300
Define symbol directive ... 300
Export directive ... 301
Expressions ... 301
Numbers ... 302
Structural configuration ... 303
If directive ... 303
Include directive ... 304

Section reference ... 305

Summary of sections ... 305
Descriptions of sections and blocks ... 307
CSTACK ... 307
.difunct ... 307
.eeprom.noinit ... 307
.far.bss ... 307
.far.data ... 308
.far.data_init ... 308
.far.noinit ... 308
.far.rodata ... 308
.far_func.text ... 309
HEAP ... 309
.huge.bss ... 309
.huge.data ... 309
.huge.data_init ... 310
.huge.noinit ... 310
.huge.rodata ... 310
.huge_func.text ... 310
.iar.dynexit ... 310
.intvec ... 311

.near.bss 311
.near.data 311
.near.data_init 311
.near.noinit 311
.near.rodata 312
.near_func.text 312
.tiny.bss 312
.tiny.data 312
.tiny.data_init 313
.tiny.noinit 313
.tiny.rodata 313
.tiny.rodata_init 313
.vregs 314

IAR utilities 315

The IAR Archive Tool—iarchive 315
Invocation syntax 315
Summary of iarchive commands 316
Summary of iarchive options 317
Diagnostic messages 317

The IAR ELF Tool—ielftool 318
Invocation syntax 319
Summary of ielftool options 319

The IAR ELF Dumper for STM8—ielfdumpstm8 320
Invocation syntax 320
Summary of ielfdumpstm8 options 321

The IAR ELF Object Tool—iobjmanip 321
Invocation syntax 321
Summary of iobjmanip options 322
Diagnostic messages 322

The IAR Absolute Symbol Exporter—isymexport 324
Invocation syntax 324
Summary of isymexport options 325
Steering files 325

Show directive 326
Hide directive 326
Rename directive 327
Diagnostic messages 327
Descriptions of options 329
--all 329
--bin 329
--checksum 330
--code 331
--create 331
--delete, -d 332
--edit 332
--extract, -x 333
-f 333
--fill 334
--ihex 334
--no_strtab 335
--output, -o 335
--ram_reserve_ranges 335
--raw 336
--remove_section 336
--rename_section 337
--rename_symbol 337
--replace, -r 338
--reserve_ranges 338
--section, -s 339
--self_reloc 339
--silent 340
--simple 340
--srec 340
--srec-len 341
--srec-s3only 341
--strip 341
--symbols 342

--toc, -t 342
--verbose, -V 342

Implementation-defined behavior 345

Descriptions of implementation-defined behavior 345
J.3.1 Translation 345
J.3.2 Environment 346
J.3.3 Identifiers 347
J.3.4 Characters 347
J.3.5 Integers 349
J.3.6 Floating point 349
J.3.7 Arrays and pointers 350
J.3.8 Hints 350
J.3.9 Structures, unions, enumerations, and bitfields 351
J.3.10 Qualifiers 351
J.3.11 Preprocessing directives 352
J.3.12 Library functions 353
J.3.13 Architecture 358
J.4 Locale 359

Index 361

Tables

1: Typographic conventions used in this guide xxxi
2: Naming conventions used in this guide xxxi
3: Data model characteristics 25
4: Memory types and their corresponding memory attributes 27
5: Code models 36
6: Function memory attributes 36
7: Sections holding initialized data 49
8: Description of a relocation error 62
9: Customizable items 69
10: Formatters for printf 70
11: Formatters for scanf 71
12: Functions with special meanings when linked with debug library 72
13: Library configurations 81
14: Descriptions of printf configuration symbols 84
15: Descriptions of scanf configuration symbols 85
16: Low-level I/O files 86
17: Example of runtime model attributes 92
18: Predefined runtime model attributes 93
19: Registers used for passing parameters 105
20: Registers used for returning values 106
21: Specifying the size of an assembler memory instruction 109
22: Call frame information resources defined in a names block 113
23: Language extensions 119
24: Section operators and their symbols 121
25: Compiler optimization levels 154
26: Compiler environment variables 169
27: ILINK environment variables 169
28: Error return codes 171
29: Compiler options summary 180
30: Linker options summary 207
31: Integer types 226

32: Floating-point types 230
33: Function pointers 231
34: Data pointers 232
35: Extended keywords summary 241
36: Pragma directives summary 249
37: Intrinsic functions summary 265
38: Predefined symbols 270
39: Traditional Standard C header files—DLIB 277
40: Embedded C++ header files 278
41: Standard template library header files 278
42: New Standard C header files—DLIB 279
43: Examples of section selector specifications 298
44: Section summary 305
45: iarchive parameters 316
46: iarchive commands summary 316
47: iarchive options summary 317
48: ielftool parameters 319
49: ielftool options summary 319
50: ielfdumpstm8 parameters 320
51: ielfdumpstm8 options summary 321
52: iobjmanip parameters 321
53: iobjmanip options summary 322
54: ielftool parameters 325
55: isymexport options summary 325
56: Message returned by strerror()—IAR DLIB library 360

Preface

Welcome to the IAR C/C++ Development Guide for STM8. The purpose of this guide is to provide you with detailed reference information that can help you to use the build tools to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

Who should read this guide

Read this guide if you plan to develop an application using the C or C++ language for the STM8 microcontroller and need detailed reference information on how to use the build tools. You should have working knowledge of:

- The architecture and instruction set of the STM8 microcontroller. Refer to the documentation from STMicroelectronics for information about the STM8 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 and linker for STM8, you should read *Part 1. Using the build tools* in this guide.

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

If you are new to using the IAR Systems build tools, we recommend that you first study the *IAR Embedded Workbench® IDE User Guide*. This guide contains a product overview, tutorials that can help you get started, conceptual and user information about the IDE and the IAR C-SPY® Debugger, and corresponding reference information.

What this guide contains

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

Part 1. Using the build tools

- *Introduction to the IAR build tools* gives an introduction to the IAR build tools, which includes an overview of the tools, the programming languages, the available device support, and extensions provided for supporting specific features of the STM8 microcontroller.
- *Developing embedded applications* gives the information you need to get started developing your embedded software using the IAR build tools.
- *Data storage* describes how to store data in memory, 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.
- *Linking using ILINK* describes the linking process using the IAR ILINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using ILINK options and tailoring the linker configuration file.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Application-related considerations* discusses a selected range of application issues related to using the compiler and linker.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Reference information

- *External interface details* provides reference information about how the compiler and linker interact with their environment—the invocation syntax, methods for

passing options to the compiler and linker, environment variables, the include file search procedure, and the different types of compiler and linker output. The chapter also describes how the diagnostic system works.

- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Linker options* gives a summary of the options, and contains detailed reference information for each linker option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Extended keywords* gives reference information about each of the STM8-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 STM8-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.
- *The linker configuration file* describes the purpose of the linker configuration file and describes its contents.
- *Section reference* gives reference information about the use of sections.
- *IAR utilities* describes the IAR utilities that handle the ELF and DWARF object formats.
- *Implementation-defined behavior* describes how the compiler handles the implementation-defined areas of the C language standard.

Other documentation

The complete set of IAR Systems development tools for the STM8 microcontroller is described in a series of guides. For information about:

- Using the IDE and the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the IAR Assembler for STM8, refer to the *IAR Assembler Reference Guide for STM8*
- Using the IAR DLIB Library functions, refer to the online help system

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

All of these guides are delivered in hypertext PDF or HTML format on the installation media. Some of them can also be delivered as printed books.

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.
- 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.]
- Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++.* Addison-Wesley.

We recommend that you visit these web sites:

- The STMicroelectronics web site, **www.st.com**, contains information and news about the STM8 microcontrollers.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

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

When referring to a directory in your product installation, for example `stm8\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.n\stm8\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, where `[]` is part of the described syntax.
{`option`**}**	A mandatory part of a command, where `{}` is part of the described syntax.
`[option]`	An optional part of a command.
`a|b|c`	Alternatives in a command.
`{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 STM8	IAR Embedded Workbench®

Table 2: Naming conventions used in this guide

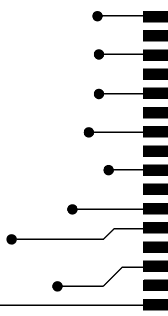
Brand name	Generic term
IAR Embedded Workbench® IDE for STM8	the IDE
IAR C-SPY® Debugger for STM8	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for STM8	the compiler
IAR Assembler™ for STM8	the assembler
IAR ILINK™ Linker	ILINK, the linker
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide (Continued)

Part 1. Using the build tools

This part of the *IAR C/C++ Development Guide for STM8* includes these chapters:

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



Introduction to the IAR build tools

This chapter gives an introduction to the IAR build tools for the STM8 microcontroller, which means you will get an overview of:

- The IAR build tools—the build interfaces, compiler, assembler, and linker
- The programming languages
- The available device support
- The extensions provided by the IAR C/C++ Compiler for STM8 to support specific features of the STM8 microcontroller.

The IAR build tools—an overview

In the IAR product installation you can find a set of tools, code examples, and user documentation, all suitable for developing software for STM8-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.

For a more detailed product overview, see the *IAR Embedded Workbench® IDE User Guide*. There you can also read about the debugger.

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

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

IAR C/C++ COMPILER

The IAR C/C++ Compiler for STM8 is a state-of-the-art optimizing compiler that offers the standard features of the C and C++ languages, plus extensions designed to take advantage of the STM8-specific facilities.

IAR ASSEMBLER

The IAR Assembler for STM8 is a powerful relocating macro assembler with a versatile set of directives and expression operators. The assembler features a built-in C language preprocessor and supports conditional assembly.

The IAR Assembler for STM8 uses the same mnemonics and operand syntax as the STM8 instruction set specified in the the *STM8 CPU Programming manual* from STMicroelectronics. For detailed information, see the *IAR Assembler Reference Guide for STM8*.

THE IAR ILINK LINKER

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

SPECIFIC ELF TOOLS

Because ILINK both uses and produces industry-standard ELF and DWARF as object format, additional IAR utilities that handle these formats can be used:

- The IAR Archive Tool—`iarchive`—creates and manipulates a library (archive) of several ELF object files
- The IAR ELF Tool—`ielftool`—performs various transformations on an ELF executable image (such as, fill, checksum, format conversion etc)
- The IAR ELF Dumper for STM8—`ielfdumpstm8`—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—`iobjmanip`—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—`isymexport`—exports absolute symbols from a ROM image file, so that they can be used when linking an add-on application.

EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IAR Embedded Workbench® IDE User Guide*.

IAR language overview

There are two high-level programming languages you can use with the IAR C/C++ Compiler for STM8:

- C, the most widely used high-level programming language in the embedded systems industry. You can build freestanding applications that follow the following standards:
  - The standard ISO/IEC 9899:1999 (including technical corrigendum No.3), also known as C99. Hereafter, this standard is referred to as *Standard C* in this guide.
  - The standard ISO 9899:1990 (including all technical corrigenda and addendum), also known as C94, C90, C89, and ANSI C. Hereafter, this standard is referred to as *C89* in this guide. This standard is required when MISRA C is enabled.
- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. IAR Systems supports two levels of the C++ language:
  - Embedded C++ (EC++), a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
  - IAR Extended Embedded C++, with additional features such as full template support, multiple inheritance, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

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

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

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *IAR Assembler Reference Guide for STM8*.

Device support

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

SUPPORTED STM8 DEVICES

The IAR C/C++ Compiler for STM8 supports all devices based on the standard STMicroelectronics STM8 microcontroller.

PRECONFIGURED SUPPORT FILES

The IAR product installation contains preconfigured files for supporting different devices.

Header files for I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. The product package supplies I/O files for all devices that are available at the time of the product release. You can find these files in the `stm8\inc` directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can be created using one of the provided ones as a template.

Linker configuration files

The `stm8\config\linker` directory contains ready-made linker configuration files for all supported devices. The files have the filename extension `icf` and contain the information required by ILINK. To read more about the linker configuration file, see *Placing code and data—the linker configuration file*, page 46, and for reference information, the chapter *The linker configuration file*.

Device description files

The debugger handles several of the device-specific requirements, such as definitions of peripheral registers and groups of these, by using device description files. These files are located in the `stm8\config\ddf` directory and they have the filename extension `ddf`. To read more about these files, see the *IAR Embedded Workbench® IDE User Guide*.

EXAMPLES FOR GETTING STARTED

The `stm8\examples` directory contains some examples of working applications to give you a smooth start with your development. Examples are provided for some of the supported devices.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the STM8 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 190 for additional information.

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

PRAGMA DIRECTIVES

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

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

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

PREDEFINED SYMBOLS

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

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

SPECIAL FUNCTION TYPES

The special hardware features of the STM8 microcontroller are supported by the compiler's special function types: interrupt, monitor, and task. You can write a complete application without having to write any of these functions in assembler language.

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 37.

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 95.

Developing embedded applications

This chapter provides the information you need to get started developing your embedded software for the STM8 microcontroller using the IAR build tools.

First, you will get an overview of the tasks related to embedded software development, followed by an overview of the build process, including the steps involved for compiling and linking an application.

Next, the program flow of an executing application is described.

Finally, you will get an overview of the basic settings needed for a project.

Developing embedded software using IAR build tools

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

MAPPING OF INTERNAL AND EXTERNAL MEMORY

Embedded systems typically contain various types of memory, such as on-chip RAM, external DRAM or SRAM, ROM, EEPROM, or flash memory.

As an embedded software developer, you must understand the features of the different memory types. For example, on-chip RAM is often faster than other types of memories, and variables that are accessed often would in time-critical applications benefit from being placed here. Conversely, some configuration data might be accessed seldom but must maintain their value after power off, so they should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. For an overview see *Controlling data and function placement in memory*, page 150. The linker places sections of code and data in memory according to the directives you specify in the linker configuration file, see *Placing code and data—the linker configuration file*, page 46.

COMMUNICATION WITH PERIPHERAL UNITS

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

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. See *Device support*, page 5. For an example, see *Accessing special function registers*, page 162.

EVENT HANDLING

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

The compiler supports the following processor exception types: reset, interrupts, and software interrupts, which means that you can write your interrupt routines in C, see *Interrupt functions*, page 37.

SYSTEM STARTUP

In all embedded systems, system startup code is executed to initialize the system—both the hardware and the software system—before the `main` function of the application is called. The CPU imposes this by starting execution from the memory location specified in the reset vector.

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

A C/C++ application further needs to initialize all global variables. This initialization is handled by the linker and the system startup code in conjunction. For more information, see *Application execution—an overview*, page 14.

REAL-TIME OPERATING SYSTEMS

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

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

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

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

The build process—an overview

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

To get familiar with the process in practice, you should run one or more of the tutorials available in the *IAR Embedded Workbench® IDE User Guide*.

THE TRANSLATION PROCESS

There are two tools in the IDE that translate application source files to intermediary object files. The IAR C/C++ Compiler and the IAR Assembler. Both produce relocatable object files in the industry-standard format ELF, including the DWARF format for debug information.

Note: The compiler can also be used for translating C/C++ source code into assembler source code. If required, you can modify the assembler source code which then can be assembled into object code. For more information about the IAR Assembler, see the *IAR Assembler Reference Guide for STM8.*

This illustration shows the translation process:

Figure 1: The build process before linking

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

THE LINKING PROCESS

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

The IAR ILINK Linker (`ilinkstm8.exe`) is used for building the final application. Normally, ILINK requires the following information as input:

- Several object files and possibly certain libraries
- A program start label (set by default)
- The linker configuration file that describes placement of code and data in the memory of the target system.

This illustration shows the linking process:

Relocatable object files
External library
Customer library
Standard C/C++ library
ILINK linker
ILINK configuration file
Map file
Absolute output ELF/DWARF

Figure 2: The linking process

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

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

For an in-depth description of the procedure performed by ILINK, see *The linking process*, page 44.

AFTER LINKING

The IAR ILINK Linker produces an absolute object file in ELF format that contains the executable image. After linking, the produced absolute executable image can be used for:

- Loading into the IAR C-SPY Debugger or any other compatible external debugger that reads ELF and DWARF.
- Programming to a flash/PROM using a flash/PROM programmer. Before this is possible, the actual bytes in the image must be converted into the standard Motorola

32-bit S-record format or the Intel Hex-32 format. For this, use `ielftool`, see *The IAR ELF Tool—ielftool*, page 318.

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

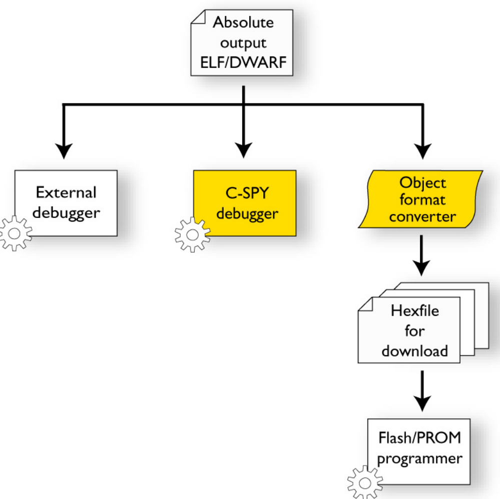


Figure 3: Possible uses of the absolute output ELF/DWARF file

Application execution—an overview

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

- Initialization phase
- Execution phase
- Termination phase.

THE INITIALIZATION PHASE

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

- Hardware initialization, which generally at least initializes the stack pointer.

  The hardware initialization is typically performed in the system startup code `cstartup.s` and if required, by an extra low-level routine that you provide. It might

include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.

- Software C/C++ system initialization

  Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.

- Application initialization

  This depends entirely on your application. It can include setting up an RTOS kernel and starting initial tasks for an RTOS-driven application. For a bare-bone application, it can include setting up various interrupts, initializing communication, initializing devices, etc.

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

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

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

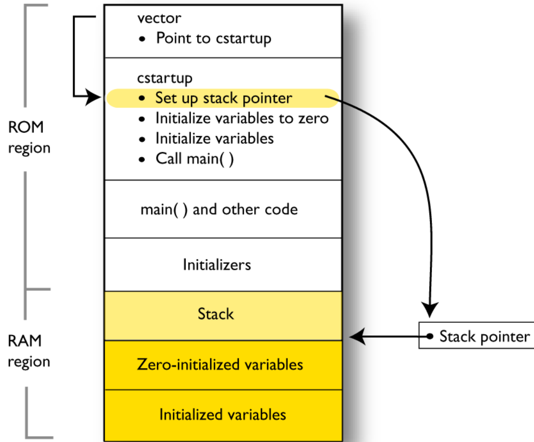


Figure 4: Initializing hardware

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

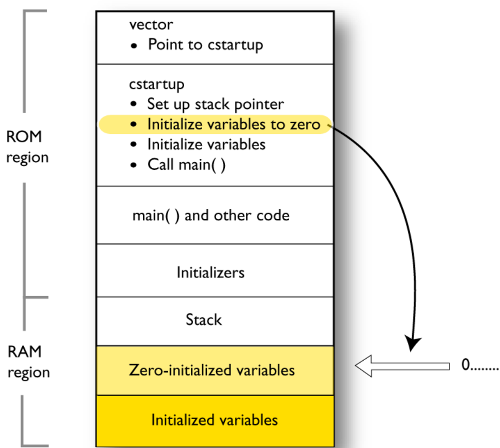


Figure 5: Zero-initializing variables

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

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

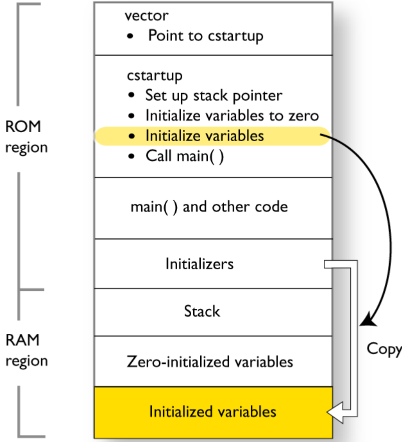


Figure 6: Initializing variables

4 Finally, the `main` function is called:

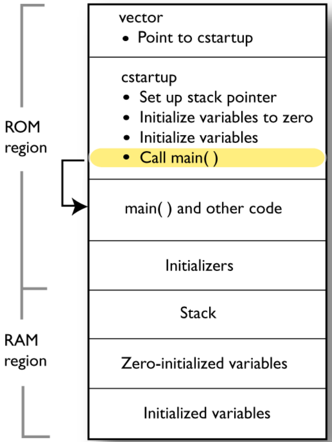


Figure 7: Calling main

For a detailed description about each stage, see *System startup and termination*, page 77. For more details about initialization of data, see *Initialization at system startup*, page 49.

THE EXECUTION PHASE

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

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

THE TERMINATION PHASE

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

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

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

To read more about this, see *System termination*, page 79.

Building applications—an overview

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

```
iccstm8 myfile.c
```

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

```
ilinkstm8 myfile.o myfile2.o -o a.out --config my_configfile.icf
```

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

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

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 STM8 device you are using. You can specify the options either from the command line interface or in the IDE.

You need to make settings for:

- Processor configuration
- Data model
- Code model
- Optimization settings
- Runtime environment
- Customizing the ILINK configuration, see the chapter *Linking your application*

In addition to these settings, many other options and settings can fine-tune the result even further. For details about how to set options and for a list of all available options, see the chapters *Compiler options*, *Linker options*, and the *IAR Embedded Workbench® IDE User Guide*, respectively.

PROCESSOR CONFIGURATION

To make the compiler generate optimum code, you should configure it for the STM8 microcontroller you are using.

Core

In the compiler, you can select an instruction set architecture for which the code will be generated.

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

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

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

DATA MODEL

One of the characteristics of the STM8 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 default memory access methods by selecting a data model. These data models are supported:

- In the *small* data model, variables are placed in the 8-bit address range
- In the *medium* data model, all data is placed in the 16-bit address range
- In the *large* data model, constants are placed in the 24-bit dress range.

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

For guidelines about the choice of data model and efficient use of memory types, see *Efficient use of memory types*, page 159.

CODE MODEL

The compiler supports *code models* that control which function calls are generated by default, which determines the size of the linked application. These code models are available:

- In the *small* code model, all functions are placed in the 16-bit address range
- In the *medium* code model, all functions are placed in the 24-bit address range. Functions are not allowed to cross 64-Kbyte section boundaries.
- In the *large* code model, all functions are placed in the 24-bit address range. Functions are allowed to cross 64-Kbyte section boundaries.

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

For guidelines about the choice of code model and efficient use of memory types, see *Efficient use of memory types*, page 159.

OPTIMIZATION FOR SPEED AND SIZE

The compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

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

RUNTIME ENVIRONMENT

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

A runtime library is provided: the IAR DLIB Library, which supports Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.

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

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

For detailed information about the runtime environment, see the chapter *The DLIB runtime environment*.

Setting up for the runtime environment in the IDE

The library is automatically chosen by the linker according to the settings you made in **Project>Options>General Options**, on the pages **Library Configuration**, **Library Options**, and **Library Usage**.

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, et cetera. See *Library configurations*, page 81, for more information.

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

Setting up for the runtime environment from the command line

You do not have to specify a library file explicitly, as ILINK automatically uses the correct library file.

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

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

```
-I stm8\inc
```

Data storage

This chapter gives a brief introduction to the memory layout of the STM8 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 STM8 microcontroller internally uses a Harvard architecture, with separate code and data memory buses. However, STM8 presents a unified 16-Mbyte linear address space to the user. Data can be accessed in the code memory and code can be fetched from the data memory.

The compiler supports STM8 devices with up to 16 Mbytes of continuous memory, ranging from `0x000000` to `0xFFFFFF`. 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. To read more about this, see *Memory types*, page 26.

DIFFERENT WAYS TO STORE DATA

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

- Auto variables.

  All variables that are local to a function, except those declared static, are stored on the stack. 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.

- 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 24 and *Memory types*, page 26.

- 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 33.

Data models

Technically, the data model specifies the default memory types. This means that the data model controls the following:

- The default placement of static and global variables, and constant literals
- The default pointer type.

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 27.

SPECIFYING A DATA MODEL

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

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 27.

This table summarizes the different data models:

Data model name	Default memory attribute for data	Default memory attribute for constants	Default pointer attribute	Placement of data	Placement of constants
Small	`__tiny`	`__near`	`__near`	The first 256 bytes of memory	The first 64 Kbytes of memory
Medium (default)	`__near`	`__near`	`__near`	The first 64 Kbytes of memory	The first 64 Kbytes of memory
Large	`__near`	`__far`	`__far`	The first 64 Kbytes of memory	The entire 16 Mbytes of memory

Table 3: Data model characteristics

See the *IAR Embedded Workbench® IDE User Guide* for information about setting options in the IDE.

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 185.

The small data model

Global variables and static local variables are placed in the first 256 bytes of memory. This allows the compiler to use smaller and faster instructions to access them directly. Constants are placed in the first 64 Kbytes of memory. The default pointer must be able to access both constants and variables, and is therefore 16 bits.

The medium data model

All data (including constants) is placed in the first 64 Kbytes of memory. The default pointer is 16 bits.

The large data model

Global variables and static local variables are placed in the first 64 Kbytes of memory. Constants are placed anywhere in the 16-MByte memory space, but are by deafult limited to at most 64 Kbytes in size.

The default pointer is 24 bits, to be able to access both variables and constants. The compiler uses larger and slower instructions for pointer accesses, but can use normal instructions for direct access to variables.

Memory types

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

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

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

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

For more information about memory access methods, see *Memory access methods*, page 109.

TINY

The tiny memory consists of the low 256 bytes of memory. In hexadecimal notation, this is the address range `0x00-0xFF`.

An object with the memory type tiny can only be placed in tiny memory, and the size of such an object is limited to 255 bytes. The compiler can use smaller and faster code for direct access to objects in tiny memory.

NEAR

The near memory consists of the low 64 Kbytes of memory. In hexadecimal notation, this is the address range `0x0000-0xFFFF`.

A near object can be placed in tiny memory or near memory, and the size of such an object is limited to 64 Kbytes-1.

FAR

Using this memory type, you can place the data objects anywhere in memory. However, the size of such an object is limited to 64 Kbytes-1, and it cannot cross a 64-Kbyte physical section boundary.

The compiler must use larger and slower code to access objects in far memory.

HUGE

An object with the memory type huge 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 compiler cannot use indexing addressing modes for objects in huge memory. Thus the code produced for accessing such objects is often larger and slower than for objects in far memory.

EEPROM

The eeprom memory consists of a part of the low 64 Kbytes of memory. In hexadecimal notation, this is somewhere in the address range `0x0000-0xFFFF`.

An eeprom object can only be placed in EEPROM memory, and the size of such an object is limited by the size of the EEPROM, or at most 64 Kbytes-1. When an assignment to an eeprom object is made, the value is automatically written to EEPROM memory.

USING DATA MEMORY ATTRIBUTES

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

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

Memory type	Keyword	Address range	Pointer size	Default in data model
Tiny	`__tiny`	0–255 bytes	8 bits	Small
Near	`__near`	0–64 Kbytes	16 bits	Medium
Far	`__far`	0–16 Mbytes	24 bits	Large
Huge	`__huge`	0–16 Mbytes	24 bits	--

Table 4: Memory types and their corresponding memory attributes

Memory type	Keyword	Address range	Pointer size	Default in data model
EEPROM	`__eeprom`	0–64 Kbytes	16 bits	--

Table 4: Memory types and their corresponding memory attributes (Continued)

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

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 190 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 242.

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 237.

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

```
__tiny int i, j;
int __tiny 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 __tiny Byte;
typedef Byte *BytePtr;
Byte aByte;
BytePtr aBytePointer;
```

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

POINTERS AND MEMORY TYPES

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

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

```
int __huge * MyPtr;
```

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

```
char __far * __tiny MyPtr2;
```

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

Differences between pointer types

A pointer must contain information needed to specify a memory location of a certain memory type. This means that the pointer sizes are different for different memory types. In the IAR C/C++ Compiler for STM8, the size of the `tiny` and `near` pointers are 8 and 16 bits, respectively. The `far` and `huge` pointers are 24 bits.

In the compiler, it is illegal, with one exception, to convert pointers between different types without explicit casts. The exception is when casting from a small pointer type to a larger pointer type.

The natural pointer size for the STM8 architecture is 16 bits. There is support for 24-bit pointers through a special load and store instruction, `LDF`. However, 24-bit pointers or addresses cannot be used as operands to any other instructions. Therefore, the code produced by the compiler for far and huge pointer accesses is larger and slower than for near accesses.

While the STM8 architecture supports direct access to tiny memory through small and fast addressing modes, there is no support for 8-bit pointers. The compiler automatically zero-extends tiny pointers to 16 bits before accessing memory using them. You should only use tiny pointers to save space in large data structures, at the expense of larger and slower code.

For more information about pointers, see *Pointer types*, page 231.

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 tiny memory.

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

__tiny struct MyStruct gamma;
```

This declaration is incorrect:

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

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in near memory is declared. The function returns a pointer to an integer in tiny 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 determined by the data model in use.
`int __near MyB;`	A variable in near memory.
`__tiny int MyC;`	A variable in tiny memory.
`int * MyD;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __near * MyE;`	A pointer stored in default memory. The pointer points to an integer in near memory.
`int __near * __tiny MyF;`	A pointer stored in tiny memory pointing to an integer stored in near memory.

`int __tiny * MyFunction(` `int __near *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in near memory. The function returns a pointer to an integer stored in tiny 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 *Classes*, page 169.

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

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

For more information about C++ classes, see *Classes*, page 169.

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—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—functions that call themselves either directly or indirectly—are used.

Dynamic memory on the heap

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

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

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

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
- 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 147. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models and memory attributes for function storage

By means of *code models*, the compiler supports placing functions in a default part of memory, or in other words, use a default function memory attribute. Technically, the code models control the default memory attribute. Indirectly, they control the following:

- The default memory range for storing the function, which implies a default memory attribute
- The maximum module size
- The maximum application size.

The compiler supports three code models. If you do not specify a code model, the compiler will use the Medium code model as default. 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.

These code models are available:

Code model name	Description
Small	Allows for up to 64 Kbytes of memory
Medium (default)	Allows for up to 16 Mbytes of memory, but functions are not allowed to cross 64-Kbyte section boundaries
Large	Allows for up to 16 Mbytes of memory

Table 5: Code models

See the *IAR Embedded Workbench® IDE User 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 183.

USING FUNCTION MEMORY ATTRIBUTES

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

Function memory attribute	Address range	Pointer size	Default in code model	Description
`__near_func`	`0x0-0xFFFF`	2 bytes	Small	Placed in the first 64 Kbytes of memory. Not allowed in the medium and large code models.
`__far_func`	`0x0-0xFFFFFF`	3 bytes	Medium	Objects cannot cross a 64-Kbyte boundary. Not allowed in the small code model.
`__huge_func`	`0x0-0xFFFFFF`	3 bytes	Large	Objects can cross a 64-Kbyte boundary. Not allowed in the small code model.

Table 6: Function memory attributes

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For details about the restrictions, see *Casting*, page 232.

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

Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for STM8 provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__interrupt`, and `__monitor`
- The pragma directive `#pragma vector`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, `__set_interrupt_state`, `__halt`, `__wait_for_exception`, and `__wait_for_interrupt`.

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 STM8 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 STM8 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 STM8 microcontroller, the interrupt vector table always starts at the address `0x8000` and is placed in the `.intvec` section. The interrupt vector is the offset into the interrupt vector table.

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

The header file `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:

```
#include <iostm8s208mb.h>

#pragma vector = UART1_R_RXNE_vector  /* Symbol from I/O */
                                      /* header file */
__interrupt void MyInterruptRoutine(void)
{
  /* Do something */
}
```

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

MONITOR FUNCTIONS

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

To define a monitor function, you can use the `__monitor` keyword. For reference information, see *__monitor*, page 244.

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.
    {
      // Interrupts are disabled while m is in scope.
```

```
      Mutex m;

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

private:
  static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
  static long nextStop = 100;

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

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, 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.
- Trap member functions cannot be declared virtual. The reason for this is that trap functions cannot be called via function pointers.

Linking using ILINK

This chapter describes the linking process using the IAR ILINK Linker and the related concepts—first with an overview and then in more detail.

Linking—an overview

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

ILINK combines one or more relocatable object files—produced by the IAR Systems compiler or assembler—with selected parts of one or more object libraries to produce an executable image in the industry-standard format *Executable and Linking Format* (ELF).

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

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

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

To handle ELF files, various tools are included. For a list of included utilities, see *Specific ELF tools*, page 4.

Modules and sections

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

- Several sections of code or data
- Runtime attributes specifying various types of information, for example the version of the runtime environment
- Optionally, debug information in DWARF format

- A symbol table of all global symbols and all external symbols used.

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

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

code	Executable code
readonly	Constant variables
readwrite	Initialized variables
zeroinit	Zero-initialized variables

Note: In addition to these section types—sections that contain the code and data that are part of your application—a final object file will contain many other types of sections, for example sections that contain debugging information or other type of meta information.

A section is the smallest linkable unit; but if possible, ILINK can exclude smaller units—section fragments—from the final application. For more information, see *Keeping modules*, page 57, and *Keeping symbols and sections*, page 57.

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

The IAR build tools have many predefined section names. See the chapter *Section reference* for more details about each section.

The linking process

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

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

- Determine which modules to include in the application. Modules provided in object files are always included. A module in a library file is only included if it provides a definition for a global symbol that is referenced from an included module.
- Select which standard library files to use. The selection is based on attributes of the included modules. These libraries are then used for satisfying any still outstanding undefined symbols.
- Determine which sections/section fragments from the included modules to include in the application. Only those sections/section fragments that are actually needed by the application are included. There are several ways to determine of which sections/section fragments that are needed, for example, the `__root` object attribute, the `#pragma required` directive, and the `keep` linker directive. In case of duplicate sections, only one is included.
- Where appropriate, arrange for the initialization of initialized variables and code in RAM. The `initialize` directive causes the linker to create extra sections to enable copying from ROM to RAM. Each section that will be initialized by copying is divided into two sections, one for the ROM part and one for the RAM part. If manual initialization is not used, the linker also arranges for the startup code to perform the initialization.
- Determine where to place each section according to the section placement directives in the *linker configuration file*. Sections that are to be initialized by copying appear twice in the matching against placement directives, once for the ROM part and once for the RAM part, with different attributes.
- Produce an absolute object file that contains the executable image and any debug information provided. The contents of each needed section in the relocatable input files is calculated using the relocation information supplied in its file and the addresses determined when placing sections. This process can result in one or more relocation failures if some of the requirements for a particular section are not met, for instance if placement resulted in the destination address for a `PC`-relative jump instruction being out of range for that instruction.
- Optionally, produce a map file that lists the result of the section placement, the address of each global symbol, and finally, a summary of memory usage for each module and library.

This illustration shows the linking process:

Figure 8: The linking process

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

Note: To see the actual content of an ELF object file, use `ielfdumpstm8`. See *The IAR ELF Dumper for STM8—ielfdumpstm8*, page 320.

Placing code and data—the linker configuration file

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

A typical linker configuration file contains definitions of:

- Available addressable memories
- Populated regions of those memories
- How to treat input sections

- Created sections
- How to place sections into the available regions.

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

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

A SIMPLE EXAMPLE OF A CONFIGURATION FILE

A simple configuration file can look like this:

```
/* The memory space denoting the maximum possible amount
   of addressable memory */
define memory Mem with size = 16M;

/* Memory regions in an address space */
define region ROM = Mem:[from 0x8000 size 0x10000];
define region RAM = Mem:[from 0x0000 size 0x1000];

/* Create a stack */
define block STACK with size = 0x200, alignment = 1 { };

/* Handle initialization */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* Initialize RW sections,
                                     exclude zero-initialized
                                     sections */

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

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

This configuration file defines one addressable memory `Mem` with the maximum of 16 Mbytes of memory. Further, it defines a ROM region and a RAM region in `Mem`, namely `ROM` and `RAM`. The ROM region has the size of 64 Kbytes, and the RAM region has the size of 4 Kbytes.

The file then creates an empty block called `STACK` with a size of 512 bytes in which the application stack will reside. To create a *block* is the basic method which you can use to get detailed control of placement, size, etc. It can be used for grouping sections, but also as in this example, to specify the size and placement of an area of memory.

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

The last part of the configuration file handles the actual placement of all the sections into the available regions. First, the interrupt vector—defined to reside in the read-only (`readonly`) section `.intvec`—is placed at the start of the `ROM` region, that is at address `0x8000`. Note that the part within `{}` is referred to as *section selection* and it selects the sections for which the directive should be applied to. Then the rest of the read-only sections are placed in the `ROM` region. Note that the section selection `{ readonly section .intvec }` takes precedence over the more generic section selection `{ readonly }`.

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

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

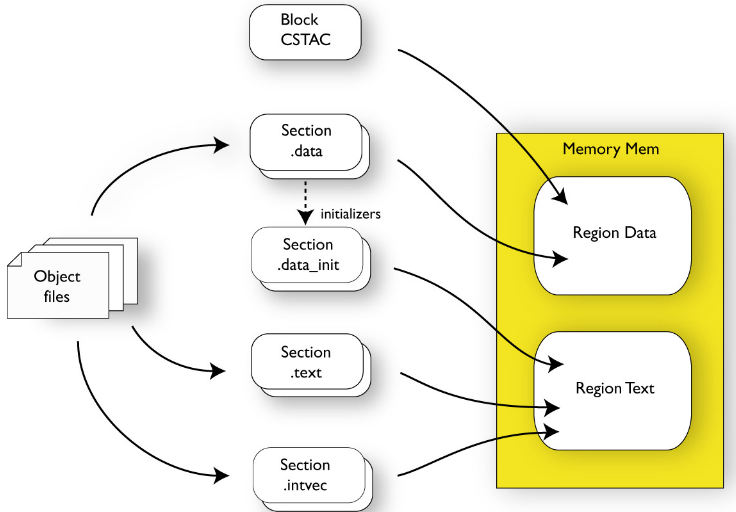


Figure 9: Application in memory

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

- Map a memory that can be addressed in multiple ways
- Handle conditional directives

- Create symbols with values that can be used in the application
- More in detail, select the sections a directive should be applied to
- More in detail, initialize code and data.

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

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

Initialization at system startup

In Standard C, all static variables—variables that are allocated at a fixed memory address—must be initialized by the runtime system to a known value at application startup. This value is either an explicit value assigned to the variable, or if no value is given, it is cleared to zero. In the compiler, there is one exception to this rule and that is variables declared `__no_init` which are not initialized at all.

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

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

Table 7: Sections holding initialized data

*** The actual memory attribute—*`memattr`*—used depends on the memory of the variable. For a list of possible section names, see *Summary of sections*, page 387.**

For a summary of all supported sections, see the chapter *Section reference*.

THE INITIALIZATION PROCESS

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

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

- Sections that should be zero-initialized are handled automatically by ILINK; they should only be placed in RAM
- Sections that should be initialized, except for zero-initialized sections, should be listed in an `initialize` directive

  Normally during linking, a section that should be initialized is split in two sections, where the original initialized section will keep the name. The contents are placed in the new initializer section, which will keep the original name suffixed with `_init`. The initializers should be placed in ROM and the initialized sections in RAM, by means of placement directives. The most common example is the `.data` section that the linker splits in `.data` and `.data_init`.
- Sections that contains constants should not be initialized; they should only be placed in flash/ROM
- Sections holding `__no_init` declared variables should not be initialized and thus should be listed in a `do not initialize` directive. They should also be placed in RAM.

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

```
/* Handle initialization */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* Initialize RW sections,
                                     exclude zero-initialized
                                     sections */

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

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

For detailed information and examples about how to configure the initialization, see *Linking considerations*, page 53.

C++ DYNAMIC INITIALIZATION

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

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

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

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

See also *Section-selectors*, page 297.

Linking your application

This chapter lists aspects that you must consider when linking your application. This includes using ILINK options and tailoring the linker configuration file.

Finally, this chapter provides some hints for troubleshooting.

Linking considerations

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

- Defining your own memory areas
- Placing sections
- Keeping modules in the application
- Keeping symbols and sections in the application
- Application startup
- Setting up the stack and heap
- Setting up the `atexit` limit
- Changing the default initialization
- Symbols for controlling the application
- Standard library handling
- Other output formats than ELF/DWARF.

CHOOSING A LINKER CONFIGURATION FILE

The `config` directory contains ready-made linker configuration files for all supported devices. The files contain the information required by ILINK. The only change you will normally have to make, if any, to the supplied configuration file is to customize the start and end addresses of each region so they fit the target system memory map. If, for example, your application uses additional external RAM, you must also add details about the external RAM memory area.

To edit a linker configuration file, use the editor in the IDE, or any other suitable editor. Alternatively, choose **Project>Options>Linker** and click the **Edit** button on the **Config** page to open the dedicated linker configuration file editor.

Remember not to change the original template file. We recommend that you make a copy in the working directory, and modify the copy instead. If you are using the linker configuration file editor in the IDE, the IDE will make a copy for you.

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

DEFINING YOUR OWN MEMORY AREAS

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

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

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

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

Each region definition must be tailored for the actual hardware.

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

Adding an additional region

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

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

Merging different areas into one region

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

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

or equivalently

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

Adding a region in a new memory

To add a region in a new memory, write:

```
/* Define a 2nd addressable memory */
define memory Mem2 with size = 64k;
/* Define a region for constants with start address 0 and 64
Kbytes large */
define region CONSTANT = Mem2:[from 0 size 0x10000];
```

Defining the unit size for a new memory

If the new memory is not byte-oriented (8-bits per byte) you should define what unit size to use:

```
/* Define the bit addressable memory */
define memory Bit with size = 256, unitbitsize = 1;
```

PLACING SECTIONS

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

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

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

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

Placing a section at a specific address in memory

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

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

Placing a section first or last in a region

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

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

Declare and place your own sections

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

```
/* Declare a section for variables. */
#pragma section = "MYOWNSECTION"

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

This is the equivalent example in assembler language:

```
            name    createSection
            section MYOWNSECTION:CONST(2)
                                        ; Create a section,
                                        ; and fill it with
            dc16    0xF0F0              ; constant bytes.
            end
```

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

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

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

RESERVING SPACE IN RAM

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

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

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

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

```
/* Declare a section for temporary storage. */
#pragma section = "TEMPSTORAGE"

char *GetTempStorageStartAddress()
{
  /* Return start address of section TEMPSTORAGE. */
```

```
  return __section_begin("TEMPSTORAGE");
}
```

KEEPING MODULES

If a module is linked as an object file, it is always kept. That is, it will contribute to the linked application. However, if a module is part of a library, it is included only if it is symbolically referred to from other parts of the application. This is true, even if the library module contains a root symbol. To assure that such a library module is always included, use `iarchive` to extract the module from the library, see *The IAR Archive Tool—iarchive*, page 315.

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

For more information about modules, see *Modules and sections*, page 43.

KEEPING SYMBOLS AND SECTIONS

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

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

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

For more information about the linking procedure for keeping symbols and sections, see *The linking process*, page 44.

APPLICATION STARTUP

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

To change the start point of the application to another label, use the ILINK option `--entry`; see *--entry*, page 213.

SETTING UP THE STACK

The size of the CSTACK block is defined in the linker configuration file. To change the allocated amount of memory, change the block definition for CSTACK:

```
define block CSTACK with size = 0x2000, alignment = 1{ };
```

Specify an appropriate size for your application.

To read more about the stack, see *Stack considerations*, page 139.

SETTING UP THE HEAP

The size of the heap is defined in the linker configuration file as a block:

```
define block HEAP with size = 0x1000, alignment = 1{ };
place in RAM {block HEAP};
```

Specify the appropriate size for your application.

SETTING UP THE ATEXIT LIMIT

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

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

CHANGING THE DEFAULT INITIALIZATION

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

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

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

Choosing packing algorithm

To override the default packing algorithm, write for example:

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

To read more about the available packing algorithms, see *Initialize directive*, page 291.

Manual initialization

The `initialize manually` directive lets you take complete control over initialization. For each involved section, ILINK creates an extra section that contains the initialization data, but makes no arrangements for the actual copying. This directive is, for example, useful for overlays:

```
/* Sections MYOVERLAY1 and MYOVERLAY2 will be overlaid in
MyOverlay */
define overlay MyOverlay { section MYOVERLAY1 };
define overlay MyOverlay { section MYOVERLAY2 };

/* Split the overlay sections but without initialization during
system startup */
initialize manually { section MYOVERLAY* };

/* Place the initializer sections in a block each */
define block MyOverlay1InRom { section MYOVERLAY1_init };
define block MyOverlay2InRom { section MYOVERLAY2_init };

/* Place the overlay and the initializers for it */
place in RAM { overlay MyOverlay };
place in ROM { block MyOverlay1InRom, block MyOverlay2InRom };
```

The application can then start a specific overlay by copying, as in this case, ROM to RAM:

```
#include <string.h>


/* Declare the overlay sections. */

#pragma section = "MYOVERLAY"
#pragma section = "MYOVERLAY1INROM"


/* Function that switches in image 1 into the overlay. */

void SwitchToOverlay1()
{
  char *targetAddr     = __section_begin("MYOVERLAY");
  char *sourceAddr     = __section_begin("MYOVERLAY1INROM");
  char *sourceAddrEnd  = __section_end("MYOVERLAY1INROM");
  int size = sourceAddrEnd - sourceAddr;

  memcpy(targetAddr, sourceAddr, size);
}
```

Running all code from RAM

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

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

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

To reduce the ROM space that is needed, it might be useful to compress the data with one of the available packing algorithms. For example,

```
initialize by copy with packing = lzw { readonly, readwrite };
```

To read more about the available compression algorithms, see *Initialize directive*, page 291.

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

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

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

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

INTERACTION BETWEEN ILINK AND THE APPLICATION

ILINK provides the command line options `--config_def` and `--define_symbol` to define symbols which can be used for controlling the application. You can also use symbols to represent the start and end of a continuous memory area that is defined in the linker configuration file. For more details, see *Interaction between the tools and your application*, page 140.

To change a reference to one symbol to another symbol, use the ILINK command line option `--redirect`. This is useful, for example, to redirect a reference from a non-implemented function to a stub function, or to choose one of several different implementations of a certain function, for example, how to choose the DLIB formatter for the standard library functions `printf` and `scanf`.

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

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

For more information, see *Interaction between the tools and your application*, page 140.

STANDARD LIBRARY HANDLING

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

To disable the automatic inclusion of the library, use the option `--no_library_search`. In this case, you must explicitly specify every library file to be included. For information about available library files, see *Using a prebuilt library*, page 67.

PRODUCING OTHER OUTPUT FORMATS THAN ELF/DWARF

ILINK can only produce an output file in the ELF/DWARF format. To convert that format into a format suitable for programming PROM/flash, use `ielftool`.

Hints for troubleshooting

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

- Messages at link time, for examples when a relocation error occurs
- The `--log` option that makes ILINK log information to `stdout`, which can be useful to understand why an executable image became the way it is, see *--log*, page 216
- The `--map` option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see *--map*, page 217.

RELOCATION ERRORS

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

A relocation error produced by ILINK can look like this:

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

The message entries are described in this table:

Message entry	Description
`Kind`	The relocation directive that failed. The directive depends on the instruction used.

Table 8: Description of a relocation error

Message entry	Description
Location	The location where the problem occurred, described with the following details: • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, 0x40000448 and "myfunc" + 0x2c. • The module, and the file. In this example, the module somecode.o. • The section number and section name. In this example, section number 7 with the name .text. • The offset, specified in number of bytes, in the section. In this example, 0x2c.
Destination	The target of the instruction, described with the following details: • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, 0x9000000c and "read" (thus, no offset). • The module, and when applicable the library. In this example, the module read.o and the library iolib.a. • The section number and section name. In this example, section number 6 with the name .text. • The offset, specified in number of bytes, in the section. In this example, 0x0.

Table 8: Description of a relocation error (Continued)

Possible solutions

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

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

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

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 `stm8\lib` and `stm8\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.
- 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 280.
- Special compiler support, for instance functions for switch handling or integer arithmetics.

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

SETTING UP THE RUNTIME ENVIRONMENT

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

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

- Choose which runtime library object file to use

  It is not necessary to specify a library file explicitly, as ILINK automatically uses the correct library file. See *Using a prebuilt library*, page 67.
- 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 81.
- 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 69. You can also specify the size and placement of the stack and the heap, see *Setting up the stack*, page 58, and *Setting up the heap*, page 58, respectively.
- Include debug support for runtime and I/O debugging

  The library offers C-SPY debug support and if you want to debug your application, you must choose to use it, see *Application debug support*, page 71
- Adapt the library functionality

  Some library functions must be customized to suit your target hardware, for example low-level functions for character-based I/O, environment functions, signal functions, and time functions. This can be done without rebuilding the entire library, see *Overriding library modules*, page 74.
- 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 sections. You do this by customizing the routine `__low_level_init`, which is executed before the data sections are initialized. See *System startup and termination*, page 77 and *Customizing system initialization*, page 80.

- 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 75.
- Check module consistency

  You can use runtime model attributes to ensure that modules are built using compatible settings, see *Checking module consistency*, page 91.

Using a prebuilt library

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

- Instruction set architecture
- Code model
- Data model
- Library configuration—Normal or Full.

CHOOSING A LIBRARY

In the IDE, the linker will include the correct library object file and library configuration file based on the options you select. See the *IAR Embedded Workbench® IDE User Guide* for additional information.

If you build your application from the command line, a default library configuration is used automatically, However, you can specify the library configuration explicitly for the compiler:

```
--dlib_config C:\...\dlstm8ssn.h
```

You can find the library object files and the library configuration files in the subdirectory `stm8\lib\`.

GROUPS OF LIBRARY FILES

The libraries are delivered in groups of library functions. The first group is:

- C/C++ standard library functions

  Contains all functions defined by Standard C and C++, for example functions like `printf` and `scanf`.

- Runtime support functions

  These are functions for system startup, initialization, floating-point arithmetics, and some of the functions that are part of Standard C and C++.

The second group is:

- Debug support functions

  These are functions for C-SPY debugging support.

Library filename syntax for C/C++ standard library functions and runtime support functions

The names of the libraries are constructed in this way:

```
dl<core><code_model><data_model><lib_config>.a
```

where

- <*core*> is stm8
- <*code_model*> is one of s, m, or l for the small, medium, and large code model, respectively
- <*data_model*> is one of s, m, or l for the small, medium, and large data model, respectively
- <*lib_config*> is one of n or f for Normal and Full, respectively.

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

Library files for debug support functions

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

```
dbg<core><code_model><data_model><debug>.a
```

where

- <*core*> is stm8
- <*code_model*> is one of s, m, or l for the small, medium, and large code model, respectively
- <*data_model*> is one of s, m, or l for the small, medium, and large data model, respectively
- <*debug*> is n for no debug and d for debug.

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 69
Startup and termination code	*System startup and termination*, page 77
Low-level input and output	*Standard streams for input and output*, page 82
File input and output	*File input and output*, page 85
Low-level environment functions	*Environment interaction*, page 88
Low-level signal functions	*Signal and raise*, page 89
Low-level time functions	*Time*, page 90
Size of heap, stack, and sections	*Stack considerations*, page 139 *Heap considerations*, page 140 *Placing code and data—the linker configuration file*, page 46

Table 9: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 74.

Choosing formatters for printf and scanf

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

Note: If you rebuild the library, you can optimize these functions even further, see *Configuration symbols for printf and scanf*, page 84.

CHOOSING PRINTF FORMATTER

The `printf` function uses a formatter called `_Printf`. The default version is quite large, and provides facilities not required in many embedded applications. To reduce the memory consumption, three smaller, alternative versions are also provided in the Standard C/EC++ library.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	Tiny	Small	Large	Full
Basic specifiers c, d, i, o, p, s, u, X, x, and %	Yes	Yes	Yes	Yes
Multibyte support	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 space, +, -, #, and 0	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 10: Formatters for printf

† **Depends on the library configuration that is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 84.

Specifying the print formatter in the IDE

To use any other formatter than the default (Small), choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.

Specifying printf formatter from the command line

To use any other formatter than the default full formatter _Printf, add one of these ILINK command line options:

```
--redirect _Printf=_PrintfTiny
--redirect _Printf=_PrintfSmall
--redirect _Printf=_PrintfLarge
```

CHOOSING SCANF FORMATTER

In a similar way to the printf function, scanf uses a common formatter, called _Scanf. The default version is very large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided in the Standard C/C++ library.

This table summarizes the capabilities of the different formatters:

Formatting capabilities	Small	Large	Full
Basic specifiers c, d, i, o, p, s, u, X, x, and %	Yes	Yes	Yes
Multibyte support	†	†	†
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 11: Formatters for scanf

† **Depends on the library configuration that is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 84.

Specifying scanf formatter in the IDE

To use any other formatter than the default (Small), choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.

Specifying scanf formatter from the command line

To use any other variant than the default full formatter `_Scanf`, add one of these ILINK command line options:

```
--redirect _Scanf=_ScanfSmall
--redirect _Scanf=_ScanfLarge
```

Application debug support

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

INCLUDING C-SPY DEBUGGING SUPPORT

You can make the library provide debugging support for:

- Handling program abort, exit, and assertions

- 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.

In the IDE, choose **Project>Options>Linker**. On the **Library** page, select the **Include C-SPY debugging support** option.

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

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

THE DEBUG LIBRARY FUNCTIONALITY

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

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

The mechanism used for implementing this feature works as follows:

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

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

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`	C-SPY notifies that the end of the application was reached
`__open`	Opens a file on the host computer

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

Function in DLIB low-level interface	Response by C-SPY
`__read`	`stdin`, `stdout`, and `stderr` will be directed 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
`__seek`	Seeks in the associated host file on the host computer
`system`	Writes a message to the Debug Log window and returns `-1`
`time`	Returns the time on the host computer
`__write`	`stdin`, `stdout`, and `stderr` will be directed to the Terminal I/O window, all other files will write to the associated host file

Table 12: Functions with special meanings when linked with debug library (Continued)

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

THE C-SPY TERMINAL I/O WINDOW

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

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

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

Speeding up terminal output

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

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

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

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

Overriding library modules

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and `low_level_init`. This can be done without rebuilding the entire library. This section describes the procedure for including your version of the module in the application project build process. The library files that you can override with your own versions are located in the `stm8\src\lib` directory.

Note: If you override a default I/O library module with your own module, C-SPY support for the module is turned off. For example, if you replace the module `__write` with your own version, the C-SPY Terminal I/O window will not be supported.

Overriding library modules using the IDE

This procedure is applicable to any source file in the library, which means that *`library_module`*`.c` in this example can be *any* module in the library.

1. Copy the appropriate *`library_module`*`.c` file to your project directory.
2. Make the required additions to the file (or create your own routine, using the default file as a model).
3. Add the customized file to your project.
4. Rebuild your project.

Overriding library modules from the command line

This procedure is applicable to any source file in the library, which means that *`library_module`*`.c` in this example can be *any* module in the library.

1. Copy the appropriate *`library_module`*`.c` to your project directory.
2. Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.
3. Compile the modified file using the same options as for the rest of the project:

```
iccstm8 library_module.c
```

This creates a replacement object module file named *`library_module`*`.o`.

Note: The core, code model, data model, include paths, and the library configuration must be the same for *`library_module`* as for the rest of your code.

4 Add *library_module*.o to the ILINK command line, either directly or by using an extended linker configuration file, for example:

```
ilinkstm8 library_module.o
```

Make sure that *library_module*.o is placed before the library on the command line. This ensures that your module is used instead of the one in the library.

Run ILINK to rebuild your application.

This will use your version of *library_module*.o, instead of the one in the library. For information about the ILINK options, see the chapter *Linker options*.

Building and using a customized library

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

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

In those cases, you must:

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

Note: To build IAR Embedded Workbench projects from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`). 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 *IAR Embedded Workbench® IDE User 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 Normal library configuration, see Table 13, *Library configurations*, page 81.

In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 19.

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities. Your library also has its own library configuration file `dlstm8`*`xyz`*`.h`, which sets up that specific library. For more information, see Table 9, *Customizable items*, page 69.

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

Modifying the library configuration file

In your library project, open the file `dlstm8`*`xyz`*`.h` and customize it by setting the values of the configuration symbols according to the application requirements.

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

USING A CUSTOMIZED LIBRARY

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

In the IDE you must do these steps:

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

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s`, `cexit.s`, and `low_level_init.c` located in the `stm8\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 80.

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:

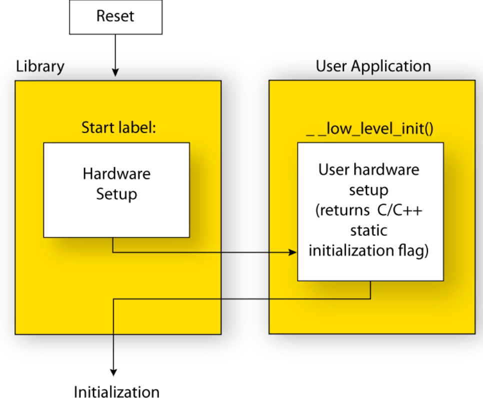


Figure 10: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__iar_program_start` in the system startup code.
- The stack pointer is initialized to the end of the `CSTACK` block
- The function `__low_level_init` is called, giving the application a chance to perform early initializations.

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

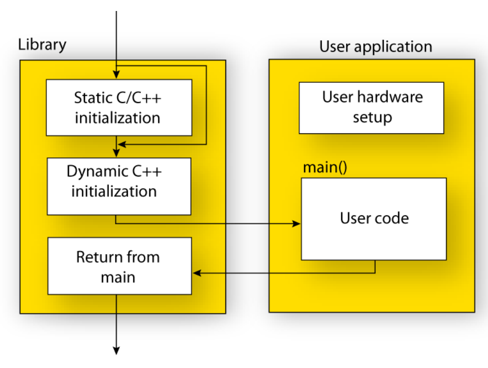


Figure 11: C/C++ initialization phase

- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more details, see *Initialization at system startup*, page 49
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

For an overview of the initialization phase, see *Application execution—an overview*, page 14.

SYSTEM TERMINATION

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

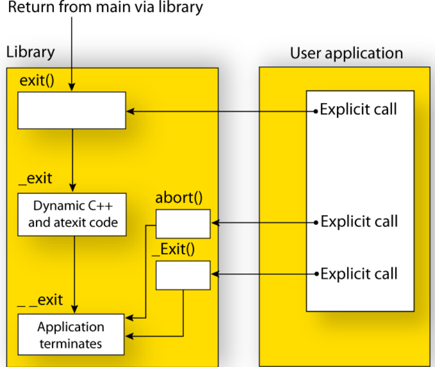


Figure 12: System termination phase

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 C 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 C-SPY debug library, 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 71.

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 sections performed by `cstartup`.

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

The code for handling system startup is located in the source files `cstartup.s` and `low_level_init.c`, located in the `stm8\src\lib` directory.

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

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 75.

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

__LOW_LEVEL_INIT

A skeleton low-level initialization file is supplied with the product: a C source file, `low_level_init.c`. The only limitation using this C source file is that static initialized variables cannot be used within the file, as variable initialization has not been performed at this point.

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

MODIFYING THE FILE CSTARTUP.S

As noted earlier, you should not modify the file `cstartup.s` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s`, 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 74.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s`. For information about how to change the start label used by the linker, see *--entry*, page 213.

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 13: Library configurations

CHOOSING A RUNTIME CONFIGURATION

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

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

- Prebuilt configuration of your choice—to specify a runtime configuration explicitly, use the `--dlib_config` compiler option. See *--dlib_config*, page 189.
- Your own configuration—you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 75.

The prebuilt libraries are based on the default configurations, see Table 13, *Library configurations*.

Standard streams for input and output

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

There are primitive I/O functions, which are the fundamental functions through which C and C++ performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these functions using whatever facilities the hardware environment provides.

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `stm8\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 75. 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 71.

Example of using __write

The code in this example uses memory-mapped I/O to write to an LCD display:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 0x50F8;

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:

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0x50F9;

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 150.

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 69.

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`

Table 14: Descriptions of printf configuration symbols

Printf configuration symbols	Includes support for
_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 14: Descriptions of printf configuration symbols (Continued)

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 15: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must;

1. Set up a library project, see *Building and using a customized library*, page 75.
2. Define the configuration symbols according to your application requirements.

File input and output

The library contains a large number of powerful functions for file I/O operations. If you use any of these functions, you must customize them to suit your hardware. To simplify adaptation to specific hardware, all I/O functions call a small set of primitive functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters.

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 81. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

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

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 16: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Application debug support*, page 71.

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 75.

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 `stm8\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 74.

THE SYSTEM FUNCTION

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 75.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For further information, see *Application debug support*, page 71.

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 `stm8\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 74.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 75.

Time

To make the `time` and `date` functions work, you must implement the three functions `clock`, `time`, and `__getzone`.

This does not require that you rebuild the library. You can find source templates in the files `clock.c` and `time.c`, and `getzone.c` in the `stm8\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 74.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 75.

The default implementation of `__getzone` specifies UTC as the time zone.

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For further information, see *Application debug support*, page 71.

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 75.

Pow

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

Assert

If you linked your application with support for runtime debugging, an assert will print a message on `stdout`. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. The `__ReportAssert` function generates the assert notification. You can find template code in the `stm8\src\lib` directory. For further information, see *Building and using a customized library*, page 75. 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 272.

Atexit

The linker allocates a static memory area for `atexit` function calls. By default, the number of calls to the `atexit` function are limited to 32. To change this limit, see *Setting up the atexit limit*, page 58.

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism that you can use to ensure that modules are built using compatible settings.

When developing an application, it is important to ensure that incompatible modules are not used together. For example, 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.

The tools provided by IAR Systems use a set of predefined runtime model attributes to automatically ensure module consistency.

RUNTIME MODEL ATTRIBUTES

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

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

Note: For IAR predefined runtime model attributes, the linker uses several ways of checking them.

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 17: Example of runtime model attributes

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

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

USING RUNTIME MODEL ATTRIBUTES

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

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

For detailed syntax information, see *rtmodel*, page 260.

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

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

For detailed syntax information, see the *IAR Assembler Reference Guide for STM8*.

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

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
`CLibrary`	`DLib`	Corresponds to the library used in the project.
`__code_model`	`small` or `medium_or_large`	Corresponds to the code model used in the project.
`__core`	`stm8`	Corresponds to the core used in the project.
`__data_model`	`small`, `medium`, or `large`	Corresponds to the data model used in the project.
`__rt_version`	*n*	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 18: Predefined runtime model attributes

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

If you are using assembler routines in the C or C++ code, refer to the chapter *Assembler directives* in the *IAR Assembler Reference Guide for STM8.*

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

For example, if you have a USART that can run in two modes, you can specify a runtime model attribute, for example `usart`. For each mode, specify a value, for example `mode1` and `mode2`. Declare this in each module that assumes that the USART is in a particular mode. This is how it could look like in one of the modules:

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

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 STM8 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 STM8 provides several ways to access low-level resources:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

This causes some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers. In many cases, the overhead of the extra instructions can be removed by the optimizer.

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

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

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

The first issue is discussed in the section *Calling assembler routines from C*, page 98. The following two are covered in the section *Calling convention*, page 101.

The section on memory access methods, page 109, covers how data in memory is accessed.

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 113.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 98, and *Calling assembler routines from C++*, page 100, respectively.

INLINE ASSEMBLER

It is possible to insert assembler code directly into a C or C++ function. The `asm` and `__asm` keywords both insert the supplied assembler statement in-line. The following example demonstrates the use of the `asm` keyword. This example also shows the risks of using inline assembler.

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

static char sFlag;

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

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

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

Inline assembler sequences have no well-defined interface with the surrounding code generated from your C or C++ code. This makes the inline assembler code fragile, and

will possibly also become a maintenance problem if you upgrade the compiler in the future. There are also several limitations to using inline assembler:

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

```
iccstm8 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 `s`. 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.s`.

Note: The `-lA` option creates a list file containing call frame information (`CFI`) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the `CFI` directives from the list file. In the IDE, 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 113.

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. 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);
  }
};
```

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
```

VIRTUAL REGISTERS

In addition to the physical registers in STM8 (`A`, `X`, `Y`, `CC`, and `SP`), the compiler also uses *virtual* registers. A virtual register is a static memory location in the fastest memory used for storing variables and temporary values.

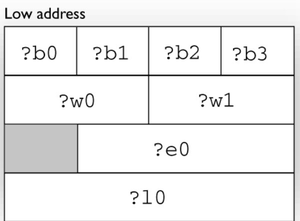
The runtime library defines 16 one-byte (8-bit) virtual registers called `?b0`, `?b1`, …, `?b15`. They are placed in the `.vregs` section, which must be allocated in RAM in the first 256 bytes of memory.

For convenience, the one-byte virtual registers are combined into larger virtual registers:

- The two-byte (16-bit) virtual registers are called `?w0`, `?w1`, …, `?w15`. They are composed as non-overlapping pairs of one-byte virtual registers, so that for example `?w1` equals `?b2:?b3`.
- The three-byte (24-bit) virtual registers are called `?e0`, `?e1`, …, `?e3`. They are composed from one-byte and two-byte virtual registers and aligned with an offset so that for example `?e1` equals `?b5:?w3`. (The offset improves the code for extending or truncating 24-bit values, because the low part can remain in the same virtual register.)
- The four-byte (32-bit) virtual registers are called `?l0`, `?l1`, `?l2`, and `?l3`. They are composed as non-overlapping pairs of two-byte virtual registers, so that for example `?l1` equals `?w2:?w3`.

Note: The combined virtual registers never cross a four-byte boundary.

This figure illustrates the virtual registers:



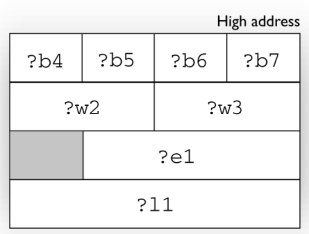


Figure 13: The first 8 bytes of virtual registers in STM8

The compiler needs the first 12 virtual registers (`?b0` to `?b11`) to generate code. Using more virtual registers can improve code size and execution speed. For details about how you can control the compiler's use of virtual registers, see --*vregs*, page 204.

To use the virtual registers in your own assembler code, include the file `vregs.inc`. This file declares labels for all the virtual registers. The labels are organized so that if you use the label for a combined register, all its sub-registers are included automatically by the linker. Be sure to follow the calling convention when you use virtual registers in your own code, because some of them must be preserved across function calls.

PRESERVED VERSUS SCRATCH REGISTERS

The general STM8 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 `A`, `X`, `Y`, `CC`, and `?b0` to `?b7` can be used as a scratch register by the function.

Preserved registers

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

The registers `?b8` through to `?b15`, are preserved registers.

Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer register must at all times point to the next free element on the stack. In the eventuality of an interrupt, everything from the point of the stack pointer and toward low memory, could be destroyed.

FUNCTION ENTRANCE

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

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

Note: Interrupt functions cannot take any parameters.

Hidden parameters

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

If the function returns a structure, the address of the memory location where the structure will be stored is passed as a hidden return value pointer.

The return value pointer is passed last.

Register parameters

These registers are available for passing parameters:

Parameters	Passed in registers
8-bit values	A, ?b0, ?b1, ?b2, ?b3, ?b4, ?b5, ?b6, ?b7
16-bit values	X, Y, ?w0, ?w1, ?w2, ?w3
24-bit values	?e0, ?e1
32-bit values	?l0, ?l1

Table 19: Registers used for passing parameters

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

Stack parameters and layout

The return address is stored in the main memory, starting at the location pointed to by the stack pointer +1. From the point of the stack pointer and toward low memory there is free space that the called function can use. The stack parameters are stored consecutively at the location of the stack pointer + 3 or 4 bytes, depending on the selected code model.

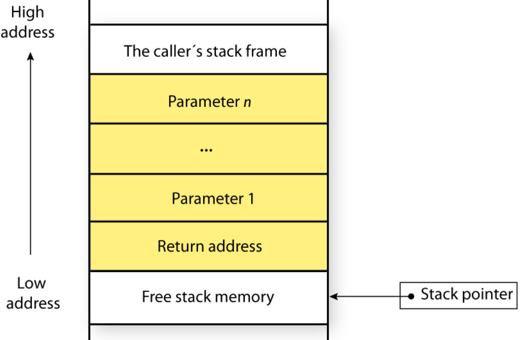


Figure 14: Stack image after the function call

FUNCTION EXIT

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

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

Registers used for returning values

The registers available for returning values are `A`, `X`, `?e0`, and `?l0`.

Return values	Passed in registers
8-bit scalar values	`A`
16-bit scalar values	`X`
24-bit scalar values	`?e0`
32-bit scalar or floating-point values	`?l0`

Table 20: Registers used for returning values

Any other values are returned using a hidden return value pointer.

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 on the stack.

The return address is restored directly from the stack:

- with the `RET` instruction for the small code model
- with the `RETF` instruction for the medium and the large code models
- with the `IRET` instruction for interrupt handlers.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

These restrictions apply:

- Functions with the type attribute `__interrupt` have only `A`, `X`, `Y`, and `CC` as scratch registers
- Functions with the object attribute `__task` do not preserve any registers
- Functions with the object attribute `__monitor` preserve the `CC` register and disable interrupts. The `CC` register is then restored at return.

To read more about type attributes, see *Type attributes*, page 237, and for more information about object attributes, see *Object attributes*, page 240.

EXAMPLES

The following section shows a series of declaration examples and the corresponding calling conventions. The complexity of the examples increases toward the end.

Example 1

Assume this function declaration:

```
int add1(int);
```

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

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    increment
            section CODE:CODE
            incw    X
            ret
            end
```

Example 2

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

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

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

The calling function must reserve 10 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register X. The return value is passed back to its caller in the register X.

Example 3

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

```
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 last parameter. The pointer to the location where the return value should be stored is passed in `Y` in the small and medium data models, and in `?e0` in the large data model. The parameter *x* is passed in `X`.

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 `X`, and the return value is returned in `X` or `?e0`, depending on the data model.

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 STM8 microcontroller.

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

Small code model

A direct call using this code model is simply:

```
        call    function
        return  RETF
```

When a function should return control to the caller, the `RET` instruction will be used.

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

```
call    (X)
```

or

```
call    (Y)
```

or, for some 16-bit virtual register:

```
call    [?w2]
```

The address is stored in a register or a virtual register and is then used for calling the function.

Medium and large code models

A direct call using this code model is simply:

```
callf   function
```

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

```
callf   [?e1]
```

The address is stored in a virtual register and is then used for calling the function.

For more information about the code models and how they differ, see *Code models and memory attributes for function storage*, page 35.

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, it will be used for explaining the reason behind the different memory types.

You should be familiar with the STM8 instruction set, in particular the different addressing modes used by the instructions that can access memory.

The IAR Assembler for STM8 uses the convention that, where ambiguous, the size of an address or offset is controlled by the prefixes `S:` and `L:`. For example:

Prefix	Size	Instruction example	Offset
S	8 bits	`ADD  A, (S:symbol,X)`	8 bits
L	16 bits	`ADD  A, (L:symbol,X)`	16 bits

Table 21: Specifying the size of an assembler memory instruction

Note that 24-bit addresses and offsets are never ambiguous, so there is no prefix for that.

For each of the access methods described in the following sections, there are three examples:

- Accessing a global variable
- Accessing a global array using an unknown index
- Accessing a structure using a pointer.

These three examples can be illustrated by this C program:

```
char MyVar;
char MyArr[10];

struct MyStruct
{
  long mA;
  char mB;
};

char Foo(int i, struct MyStruct *p)
{
  return MyVar + MyArr[i] + p->mB;
}
```

THE TINY MEMORY ACCESS METHOD

Tiny memory is located in the first 256 bytes of memory. This is the only memory type that can be accessed using 8-bit pointers and using an 8-bit index type. The advantage is that direct accesses can use smaller and faster addressing modes, and that pointers require less space. As you can see, array indexing and pointer accesses to tiny memory is not efficient.

Examples

These examples access tiny memory in different ways:

Code	Description
`ld a, s:MyVar`	Access the global variable `MyVar`

`clrw x`
`ld xl, a`
`ld a,(s:MyArr,x)`
	Access an entry in the global array `MyArr`

`clrw x`
`ld xl, a`
`ld a, (4,x)`
	Access through a pointer

THE NEAR MEMORY ACCESS METHOD

Near memory is located in the first 64 Kbytes of memory, which also includes tiny memory. Most instructions have natural addressing modes for near memory operands.

Both the pointer and the index type have a size of 16 bits.

Examples

These examples access near memory in different ways:

Code		Description
`ld`	`a, l:MyVar`	Access the global variable `MyVar`
`ld`	`a, (l:MyArr,x)`	Access the array `MyArr`
`ld`	`a, (4,x)`	Access through a pointer

THE FAR MEMORY ACCESS METHOD

The far memory access method can access the entire 16-Mbyte memory range. Far memory includes the near and tiny memories. Objects have a maximum size of 64 Kbytes and are not allowed to cross 64-Kbyte section boundaries.

The pointer has a size of 24 bits and the index type has a size of 16 bits.

The drawback of this access method is that it is only supported by the `LDF` instruction for load and store. Therefore, no other register-memory instructions can be used in far memory.

Examples

These examples access far memory in different ways:

Code		Description
`ldf`	`a, MyVar`	Access the global variable `MyVar`
`ldf`	`a, (MyArr,x)`	Access the array `MyArr`
`ldw` `ldf`	`x, #4` `a, ([?e0.e],x)`	Access through a pointer

THE HUGE MEMORY ACCESS METHOD

The huge memory access method can access the same 16-Mbyte memory range as the far method. However, there is no limit on the size or placement of objects in huge memory.

In addition to the drawbacks of the far memory access method, any indexing operations on huge memory addresses must be computed outside the instruction. This leads to large and slow code.

The pointer has a size of 24 bits and the index type has a size of 32 bits.

Examples

These examples access huge memory in different ways:

Code	Description
`ldf a, MyVar`	Access the global variable `MyVar`

`call ?sext32_l0_x`
`ldw x, LWRD(MyArr)`
`ldw s:?l1+2, x`
`ldw x, HWRD(MyArr)`
`ldw s:?l1, x`
`call ?add32_l0_l0_l1`
`ldf a, [?e0.e]`
	Access an array

`clr s:?l0`
`clrw x`
`ldw s:?l1, x`
`ldw x, #4`
`ldw s:?l1+2, x`
`call ?add32_l0_l0_l1`
`ldf a, [?e0.e]`
	Access through a pointer

THE EEPROM MEMORY ACCESS METHOD

The eeprom memory access method can access the EEPROM memory, which is a part of the first 64 Kbytes of memory. This memory access method automatically writes to the EEPROM memory when an assignment is made.

The pointer has a size of 16 bits and the index type has a size of 16 bits.

Examples

These examples access the EEPROM memory in different ways:

Code	Description
`ld a, l:MyVar`	Access the global variable `MyVar`
`ld a, (l:MyArr,x)`	Access the array `MyArr`
`ld a, (4,x)`	Access through a pointer

Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *IAR Assembler Reference Guide for STM8*.

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
`cfiNames0`	CFI names
`CFA`, `SP`, `DATA`	The CFI stack frame
`SP`	The stack pointer
`A:8`, `XL:8`, `XH:8`, `YL:8`, `YH:8`, `SP:16`, `CC:8`	Physical registers
`?RET16:16`	CFI virtual resource. The return address in the small code model.
`?RET24:24`	CFI virtual resource. The return address in the medium and large code models.

Table 22: Call frame information resources defined in a names block

Resource	Description
?b0:8–?b15:8	Virtual registers

Table 22: Call frame information resources defined in a names block (Continued)

CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

The recommended way to create an assembler language routine that handles call frame information correctly is to start with an assembler language source file created by the compiler.

1 Start with suitable C source code, for example:

```
int F(int);
int cfiExample(int i)
{
  return i + F(i);
}
```

2 Compile the C source code, and make sure to create a list file that contains call frame information—the CFI directives.

On the command line, use the option `-lA`.

In the IDE, choose **Project>Options>C/C++ Compiler>List** and make sure the suboption **Include call frame information** is selected.

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

```
NAME Cfi

        RTMODEL "__SystemLibrary", "DLib"
        RTMODEL "__code_model", "small"
        RTMODEL "__core", "stm8"
        RTMODEL "__data_model", "medium"
        RTMODEL "__rt_version", "2"

        EXTERN ?w4
        EXTERN F
        EXTERN ?epilogue_?w4
        EXTERN ?push_?w4

        PUBLIC cfiExample

        CFI Names cfiNames0
        CFI StackFrame CFA SP DATA
        CFI Resource A:8, XL:8, XH:8, YL:8, YH:8, SP:16, CC:8
        CFI VirtualResource ?RET16:16, ?RET24:24
        CFI Resource ?b0:8, ?b1:8, ?b2:8, ?b3:8, ?b4:8, ?b5:8
        CFI Resource ?b6:8, ?b7:8, ?b8:8, ?b9:8, ?b10:8
```

```
        CFI Resource ?b11:8, ?b12:8, ?b13:8, ?b14:8, ?b15:8
        CFI EndNames cfiNames0

        CFI Common cfiCommon0 Using cfiNames0
        CFI CodeAlign 1
        CFI DataAlign 1
        CFI ReturnAddress ?RET16 CODE
        CFI CFA SP+2
        CFI A Undefined
        CFI XL Undefined
        CFI XH Undefined
        CFI YL Undefined
        CFI YH Undefined
        CFI CC Undefined
        CFI ?RET16 Frame(CFA, -1)
        CFI ?RET24 Undefined
        CFI ?b0 Undefined
        CFI ?b1 Undefined
        CFI ?b2 Undefined
        CFI ?b3 Undefined
        CFI ?b4 Undefined
        CFI ?b5 Undefined
        CFI ?b6 Undefined
        CFI ?b7 Undefined
        CFI ?b8 SameValue
        CFI ?b9 SameValue
        CFI ?b10 SameValue
        CFI ?b11 SameValue
        CFI ?b12 SameValue
        CFI ?b13 SameValue
        CFI ?b14 SameValue
        CFI ?b15 SameValue
        CFI EndCommon cfiCommon0


        SECTION `.near_func.text`:CODE:NOROOT(0)
cfiExample:
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample
        CALL      L:?push_?w4
        CFI ?b9 Frame(CFA, -2)
        CFI ?b8 Frame(CFA, -3)
        CFI CFA SP+4
        LDW       S:?w4, X
        CALL      L:F
        ADDW      X, L:?w4
        JP        L:?epilogue_?w4
```

```
CFI EndBlock cfiBlock0

SECTION VREGS:DATA:NOROOT(0)

END
```

Note: The header file `cfi.m` contains the macros `XCFI_NAMES`, `XCFI_COMMON_SMALL`, and `XCFI_COMMON_MEDIUM_LARGE` which declare a typical names block and typical common blocks for different code models. These three 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 STM8 supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

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

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

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

Note: Even though it is a C99 feature, the IAR C/C++ Compiler for STM8 does not support UCNs (universal character names).

Inline assembler

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

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

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

The syntax is:

```
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"
     "            jp Label");
```

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

For more information about inline assembler, see *Mixing C and assembler*, page 95.

Extensions overview

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

This is an overview of the available extensions:

- IAR C language extensions

  For a summary of available language extensions, see *IAR C language extensions*, page 120. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

  The `#pragma` directive is defined by Standard C and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For a list of available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

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

- Intrinsic functions

  The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 95. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

  The IAR DLIB Library provides the C and C++ library definitions that apply to embedded systems. For more information, see *IAR DLIB Library*, page 276.

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 a list of extensions, see *IAR C language extensions*, page 120.
`-e`	**Standard with IAR extensions**	All *IAR C language extensions* are enabled.

Table 23: 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 122.

EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

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

- Memory attributes, type attributes, and object attributes

  For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.
- Placement at an absolute address or in a named section

  The `@` operator or the directive `#pragma location` can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named section. For more information about using these features, see *Controlling data and function placement in memory*, page 150, and *location*, page 257.
- Alignment control

  Each data type has its own alignment; for more details, see *Alignment*, page 225. If you want to change the alignment, the `#pragma data_alignment` directive is 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 148.

- 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 227.

Dedicated section operators

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

These operators behave syntactically as if declared like:

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

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

The `__section_begin` operator returns the address of the first byte of the named section or block.

The `__section_end` operator returns the address of the first byte *after* the named *`section`* or block.

The `__section_size` operator returns the size of the named section or block in bytes.

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

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

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

.

Operator	Symbol
`__section_begin(sec)`	`sec$$Base`
`__section_end(sec)`	`sec$$Limit`

Table 24: Section operators and their symbols

Operator	Symbol
`__section_size(sec)`	`sec$$Length`

Table 24: Section operators and their symbols (Continued)

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

Example

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

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

See also *section*, page 261, and *location*, page 257.

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 232.
- 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, with the function name as context. 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 190.

- 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++

The IAR C/C++ Compiler for STM8 supports two levels of the C++ language: The industry-standard Embedded C++ and IAR Extended Embedded C++. They are described in this chapter.

Overview

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

STANDARD EMBEDDED C++

The following C++ features are supported:

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

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

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

- Templates
- Multiple and virtual inheritance
- Exception handling

- 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 C++ SUPPORT

In the compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See *--ec++*, page 190.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See *--eec++*, page 190.

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

Feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for STM8, you must be aware of some benefits and some possible quirks when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

CLASSES

A class type `class` and `struct` in C++ can have static and non-static data members, and static and non-static function members. The non-static function members can be further divided into virtual function members, non-virtual function members, constructors, and destructors. For the static data members, static function members, and non-static non-virtual function members the same rules apply as for statically linked symbols outside of a class. In other words, they can have any applicable IAR-specific type, memory, and object attribute.

The non-static virtual function members can have any applicable IAR-specific type, memory, and object attribute as long as a pointer to the member function can be implicitly converted to the default function pointer type. The constructors, destructors, and non-static data members cannot have any IAR attributes.

The location operator `@` can be used on static data members and on any type of function members.

For further information about attributes, see *Type qualifiers*, page 234.

Example

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

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

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

```
  // Locate a virtual function in __far_func memory
  virtual __far_func void H();

  // Locate a virtual function into SPECIAL
  virtual void M() const volatile @ "SPECIAL";
};
```

The this pointer

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

Class memory

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

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

Example

```
class __near C
{
public:
  void MyF();       // Has a this pointer of type C __near *
  void MyF() const; // Has a this pointer of type
                    // C __near const *
  C();              // Has a this pointer pointing into near
                    // memory
  C(C const &);     // Takes a parameter of type C __near
                    // const & (also true of generated copy
                    // constructor)
  int mI;
};
```

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

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

C *Cp1;              // Creates a pointer to near memory
C __tiny *Cp2;       // Creates a pointer to tiny memory
```

Note: To place the C class into far or huge memory is not allowed because a `__far` or `__huge` pointer cannot be implicitly converted into a `__near` pointer.

Whenever a class type associated with a class memory type, like `C`, must be declared, the class memory type must be mentioned as well:

```
class __far C;
```

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

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

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

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

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

class F : public C
```

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

Note that the following is not allowed because far (or huge) is not inside near memory:

```
class __far G:public C
{
};
```

For more information about memory types, see *Memory types*, page 26.

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
}
```

TEMPLATES

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

Templates and data memory attributes

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

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

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

Example

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

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

In Extended Embedded C++, the function template parameter deduction algorithm works like this:

When function template matching is performed and an argument is used for the deduction; if that argument is a pointer to a memory that can be implicitly converted to a default pointer, do the parameter deduction as if it was a default pointer.

When an argument is matched against a reference, do the deduction as if the argument and the parameter were both pointers.

Example

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

void MyF()
{
  fun((int __near *) 0);    // T = int. The result is different
                            // than the analogous situation with
                            // class template specializations.
  fun((int        *) 0);    // T = int
  fun((int __far  *) 0);    // T = int
```

```
  fun((int __huge *) 0);    // T = int __huge
}
```

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to “small” memory types. For “large” and “other” memory types (memory that cannot be pointed to by a default pointer) it is possible. To make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

Example

```
// We assume that __far is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);

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

The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in *Extended Embedded C++*, page 128.

STL and the IAR C-SPY® Debugger

C-SPY has built-in display support for the STL containers. The logical structure of containers is presented in the watch views in a comprehensive way that is easy to understand and follow.

To read more about displaying STL containers in the C-SPY debugger, see the *IAR Embedded Workbench® IDE User Guide*.

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.

POINTER TO MEMBER FUNCTIONS

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a member function, make sure that all functions that should be pointed to reside in the default memory or a memory contained in the default memory.

Example

```
class X
{
public:
  __near_func void F();
};

void (__near_func X::*PMF)(void) = &X::F;
```

USING INTERRUPTS AND EC++ DESTRUCTORS

If interrupts are enabled and the interrupt functions use static class objects that need to be destroyed (using destructors), there might be problems if the interrupt occur during or after application exits. If an interrupt occurs after the static class object was destroyed, the application will not work properly.

To avoid this, make sure that interrupts are disabled when returning from main or when calling exit or abort. To do this, call the intrinsic function __disable_interrupt.

C++ language extensions

When you use the compiler in C++ mode and enable IAR language extensions, the following C++ language extensions are available in the compiler:

- In a `friend` declaration of a class, the `class` keyword can be omitted, for example:

```
class B;
class A
{
  friend B;       //Possible when using IAR language
                  //extensions
  friend class B; //According to standard
};
```

- Constants of a scalar type can be defined within classes, for example:

```
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 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 standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.

Note: If you use any of these constructions without first enabling language extensions, errors are issued.

Application-related considerations

This chapter discusses a selected range of application issues related to developing your embedded application.

Typically, this chapter highlights issues that are not specifically related to only the compiler or the linker.

Output format considerations

The linker produces an absolute executable image in the ELF/DWARF object file format.

You can use the IAR ELF Tool—`ielftool`— to convert an absolute ELF image to a format more suitable for loading directly to memory, or burning to a PROM or flash memory etc.

`ielftool` can produce these output formats:

- Plain binary
- Motorola S-records
- Intel hex.

Note: `ielftool` can also be used for other types of transformations, such as filling and calculating checksums in the absolute image.

The source code for `ielftool` is provided in the `arm/src` directory. For more information about `ielftool`, see *The IAR ELF Tool—ielftool*, page 318.

Stack considerations

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

The data section used for holding the stack is called `CSTACK`. The system startup code initializes the stack pointer to the end of the stack.

STACK SIZE CONSIDERATIONS

The compiler uses the internal data stack, `CSTACK`, for a variety of user application operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too large, RAM will be wasted. If the given stack size is too small, variable storage will be overwritten, leading to undefined behavior.

For more information about the stack size, see *Setting up the stack*, page 58, and *Saving stack space and RAM memory*, page 160. For information about how to customize the stack size, see the *IAR Embedded Workbench® IDE User Guide*. See also the chip manufacturer's documentation for details about stack size.

Heap considerations

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

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

- Linker sections used for the heap
- Allocating the heap size, see *Setting up the heap*, page 58.

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

Heap size and standard I/O

If you excluded `FILE` descriptors from the DLIB runtime environment, as in the normal configuration, there are no input and output buffers at all. Otherwise, as in the full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY® Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an STM8 microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

Interaction between the tools and your application

The linking process and the application can interact symbolically in four ways:

- Creating a symbol by using the ILINK command line option `--define_symbol`. ILINK will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, et cetera.

- Creating an exported configuration symbol by using the command line option `--config_def` or the configuration directive `define symbol`, and exporting the symbol using the `export symbol` directive. ILINK will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.

  One advantage of this symbol definition is that this symbol can also be used in expressions in the configuration file, for example to control the placement of sections into memory ranges.
- Using the compiler operators `__section_begin`, `__section_end`, or `__section_size`, or the assembler operators `SFB`, `SFE`, or `SIZEOF` on a named section or block. These operators provide access to the start address, end address, and size of a contiguous sequence of sections with the same name, or of a linker block specified in the linker configuration file.
- The command line option `--entry` informs ILINK about the start label of the application. It is used by ILINK as a root symbol and to inform the debugger where to start execution.

The following lines illustrate how to use these mechanisms. Add these options to your command line:

```
--define_symbol NrOfElements=10
--config_def HeapSize=1024
```

The linker configuration file can look like this:

```
define memory Mem with size = 16M;
define region ROM = Mem:[from 0x8000 size 0x10000];
define region RAM = Mem:[from 0x0000 size 0x1000];

/* Export of symbol */
export symbol HeapSize;

/* Setup a heap area with a size defined by an ILINK option */
define block MyHEAP with size = HeapSize {};

place in RAM { block MyHEAP };
```

Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by ILINK option to dynamically allocate an
array of
 * elements with specified size. The value takes the form of a
label.
 */
extern int NrOfElements;

typedef char Elements;
Elements *GetElementArray()
{
  return malloc(sizeof(Elements) * (long) &NrOfElements);
}

/* Use a symbol defined by ILINK option, a symbol that in the
 * configuration file was made available to the application.
 */
extern char HeapSize;

/* Declare the section that contains the heap. */
#pragma section = "MYHEAP"

char *MyHeap()
{
  /* First get start of statically allocated section, */
  char *p = __section_begin("MYHEAP");

  /* ...then we zero it, using the imported size. */
  for (int i = 0; i < (int) &HeapSize; ++i)
  {
    p[i] = 0;
  }
  return p;
}
```

Checksum calculation

The IAR ELF Tool—`ielftool`—fills specific ranges of memory with a pattern and then calculates a checksum for those ranges. The calculated checksum replaces the value of an existing symbol in the input ELF image. The application can then verify that the ranges did not change.

To use checksumming to verify the integrity of your application, you must:

- Reserve a place, with an associated name and size, for the checksum calculated by `ielftool`
- Choose a checksum algorithm, set up `ielftool` for it, and include source code for the algorithm in your application
- Decide what memory ranges to verify and set up both `ielftool` and the source code for it in your application source code.

Note: To set up `ielftool` in the IDE, choose **Project>Options>Linker>Checksum**.

CALCULATING A CHECKSUM

In this example, a checksum is calculated for ROM memory at `0x8002` up to `0x8FFF` and the 2-byte calculated checksum is placed at `0x8000`.

Creating a place for the calculated checksum

You can create a place for the calculated checksum in two ways; by creating a global C/C++ or assembler constant symbol with a proper size, residing in a specific section (in this example `.checksum`), or by using the linker option `--place_holder`.

For example, to create a 2-byte space for the symbol `__checksum` in the section `.checksum`, with alignment 4:

```
--place_holder __checksum,2,.checksum,4
```

To place the `.checksum` section, you must modify the linker configuration file. It can look like this (note the handling of the block `CHECKSUM`):

```
define memory Mem with size = 4M;

define region ROM_region = Mem:[from 0x8000 to 0xFFFF];

define region RAM_region = Mem:[from 0x0000 to 0x17FF];

initialize by copy { rw };
do not initialize { section .*.noinit };

define block HEAP    with alignment = 8, size = 1K{};
define block CSTACK  with alignment = 8, size = 1K{};

define block CHECKSUM { ro section .checksum };

// Placement
place at address Mem:0x0 { ro section .intvec};
place in ROM_region      { ro, first block CHECKSUM };
place in RAM_region      { rw, block HEAP, block CSTACK };
```

Running ielftool

To calculate the checksum, run `ielftool`:

```
ielftool --fill=0x00;0x8000-0x8FFF
--checksum=__checksum:2,crc16;0x8000-0x8FFF sourceFile.out
destinationFile.out
```

To calculate a checksum you also must define a fill operation. In this example, the fill pattern `0x0` is used. The checksum algorithm used is crc16.

Note that `ielftool` needs an unstripped input ELF image. If you use the `--strip` linker option, remove it and use the `--strip` ielftool option instead.

ADDING A CHECKSUM FUNCTION TO YOUR SOURCE CODE

To check the value of the `ielftool` generated checksum, it must be compared with a checksum that your application calculated. This means that you must add a function for checksum calculation (that uses the same algorithm as `ielftool`) to your application source code. Your application must also include a call to this function.

A function for checksum calculation

This function—a slow variant but with small memory footprint—uses the crc16 algorithm:

```
unsigned short slow_crc16(unsigned short sum,
                          unsigned char *p,
                          unsigned int len)
{
  while (len--)
  {
    int i;
    unsigned char byte = *(p++);

    for (i = 0; i < 8; ++i)
    {
      unsigned long oSum = sum;
      sum <<= 1;
      if (byte & 0x80)
        sum |= 1;
      if (oSum & 0x8000)
        sum ^= 0x1021;
      byte <<= 1;
    }
  }
  return sum;
}
```

You can find the source code for the checksum algorithms in the `stm8\src\linker` directory of your product installation.

Checksum calculation

This code gives an example of how the checksum can be calculated:

```
/* Start and end of the checksum range */
unsigned long ChecksumStart = 0x8000+2;
unsigned long ChecksumEnd   = 0x8FFF;

/* The checksum calculated by ielftool
 * (note that it lies on address 0x8000)
 */
extern unsigned short const __checksum;


void TestChecksum()
{
  unsigned short calc = 0;
  unsigned char zeros[2] = {0, 0};

  /* Run the checksum algorithm */
  calc = slow_crc16(0,
                    (unsigned char *) ChecksumStart,
                    (ChecksumEnd - ChecksumStart+1));

  /* Rotate out the answer */
  calc = slow_crc16(calc, zeros, 2);

  /* Test the checksum */
  if (calc != __checksum)
  {
    abort();   /* Failure */
  }
}
```

THINGS TO REMEMBER

When calculating a checksum, you must remember that:

- The checksum must be calculated from the lowest to the highest address for every memory range
- Each memory range must be verified in the same order as defined
- It is OK to have several ranges for one checksum
- If several checksums are used, you should place them in sections with unique names and use unique symbol names

- If a slow function is used, you must make a final call to the checksum calculation with as many bytes (with the value `0x00`) as there are bytes in the checksum.

For more information, see also *The IAR ELF Tool—ielftool*, page 318.

C-SPY CONSIDERATIONS

By default, a symbol that you have allocated in memory by using the linker option `--place_holder` is considered by C-SPY to be of the type `int`. If the size of the checksum is less than four bytes, you can change the display format of the checksum symbol to match its size.

In the C-SPY Watch window, select the symbol and choose **Show As** from the context menu. Choose the display format that matches the size of the checksum symbol.

Efficient coding for embedded applications

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

The topics discussed are:

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

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

Selecting data types

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

USING EFFICIENT DATA TYPES

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

- Use small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- Avoid 32-bit data types. that is `long` and `float`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- When using arrays, it is usually more efficient if the type of the index expression matches the index type of the memory of the array. For tiny this is `signed char`,

for near, eeprom, and far this is `int`, and for huge it is `long`. However, because the natural size for pointers and indexing on the STM8 architecture is 16 bits, it can sometimes be more efficient to use 16-bit index expressions for arrays in tiny memory.

- Using floating-point types on a microprocessor without a math co-processor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer parameter to point to `const` data might open for better optimizations in the calling function.

For details about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler only supports the 32-bit floating-point format. The 64-bit floating-point format is not supported. The `double` type will be treated as a `float` type.

For more information about floating-point types, see *Floating-point types*, page 230.

USING THE BEST POINTER TYPE

The natural pointer size for STM8 is 16 bits. 24-bit pointers cannot be used in register-memory operations, which leads to inefficient code. 8-bit pointers save space in variables and structures, but must be implicitly converted to 16 bits before use, which means they are also inefficient. Use 16-bit pointers whenever you can.

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 STM8 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 190, 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;
  };
} @ 0x1234;


/* The variables are used here.*/

void Test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

This declares an I/O register byte `IOPORT` at address `0x1234`. 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. To read more about data and code models, see *Data models*, page 24, and *Code models and memory attributes for function storage*, page 35, respectively.
- Memory attributes

  Use memory attributes to override the default addressing mode and placement of individual functions and data objects. To read more about memory attributes for data and functions, see *Using data memory attributes*, page 27, and *Using function memory attributes*, page 36, respectively.
- 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 `__no_init`. This is useful for individual data objects that must be located at a fixed address, for example variables with external requirements. Note that it is not possible to use this notation for absolute placement of individual functions.
- The `@` operator and the `#pragma location` directive for section placement

  Use the `@` operator or the `#pragma location` directive to place groups of functions or global and static variables in named sections, without having explicit control of each object. The sections can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the section 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 sections 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 sections as described in *Modules and sections*, page 43. At link time, one of the most important functions of the linker is to assign load addresses to the various sections used by the application. All

sections, except for the sections holding absolute located data, are automatically allocated to memory according to the specifications in the linker configuration file, as described in *Placing code and data—the linker configuration file*, page 46.

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).

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.

Note: A variable placed in an absolute location should be defined in an include file, to be included in every module that uses the variable. An unused definition in a module will be ignored. A normal `extern` declaration—one that does not use an absolute placement directive—can refer to a variable at an absolute address; however, optimizations based on the knowledge of the absolute address cannot be performed.

Examples

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

```
__no_init volatile char alpha @ 0xFF;    /* OK */
```

This example contains a `const` declared object which is not initialized. The object is placed in ROM. This is useful for configuration parameters, which are accessible from an external interface.

```
#pragma location=0x8080
__no_init const int beta;                 /* OK */
```

The actual value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

This example shows incorrect usage:

```
int delta @ 0x8086;                /* Error, not __no_init */
```

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 SECTIONS

This method can be used for placing data or functions in named sections other than default: The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named sections. The named section can either be a predefined section, or a user-defined section.

C++ static member variables can be placed in named sections just like any other static variable.

If you use your own sections, in addition to the predefined sections, the sections must also be defined in the linker configuration file.

Note: Take care when explicitly placing a variable or function in a predefined section other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

The location of the sections can be controlled from the linker configuration file.

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

Examples of placing variables in named sections

In the following examples, a data object is placed in a user-defined section. The variable will be treated as if it is located in the default memory. Note that you must place the section accordingly in the linker configuration file.

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

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

As usual, you can use memory attributes to direct the variable to a non-default memory (and then also place the section accordingly in the linker configuration file):

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

Examples of placing functions in named sections

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

#pragma location="MY_FUNCTIONS"
void h(void);
```

Specify a memory attribute to direct the function to a specific memory, and then modify the segment placement in the linker configuration file accordingly:

```
__near_func void f(void) @ "MY_NEAR_FUNC_FUNCTIONS";
```

Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to *optimize*, page 258, 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 193.

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 188.

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 Common subexpression elimination

Table 25: Compiler optimization levels

Optimization level	Description
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 25: Compiler optimization levels (Continued)

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 155.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--no_inline`, alternatively its equivalent in the IDE **Function inlining**, or the `#pragma optimize` directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis

- Cross call.

Common subexpression elimination

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

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

To read more about the command line option, see --*no_cse*, page 194.

Loop unrolling

It is possible to duplicate the loop body of a small loop, whose number of iterations can be determined at compile time, to reduce the loop overhead.

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

To read more about the command line option, see --*no_unroll*, page 197.

Function inlining

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level **High**, normally reduces execution time, but the resulting code might be difficult to debug.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed. Normally, code size does not increase when optimizing for size. To control the heuristics for individual functions, use the `#pragma inline` directive or the Standard C `inline` keyword.

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

To read more about the command line option, see --*no_inline*, page 247. For information about the pragma directive, see *inline*, page 318.

Code motion

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

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

Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object are performed using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers can reference the same memory location or not.

Type-based alias analysis is performed at optimization level **High**. For application code conforming to standard C or C++ application code, this optimization can reduce code size and execution time. However, non-standard C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

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

To read more about the command line option, see --*no_tbaa*, page 196.

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.

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.

To read more about related command line options, see *--no_cross_call*, page 194.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code, for example:

- Using efficient addressing modes
- Helping the compiler optimize
- Generating more useful error message.

WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

The following is a list of programming techniques that will, when followed, enable the compiler to better optimize the application.

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 156. 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 154.

- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more details, see *Mixing C and assembler*, page 95.

EFFICIENT USE OF MEMORY TYPES

Using the appropriate memory types is important to generate efficient code on STM8. The default memory types for functions and data are controlled by the code and data model, respectively. It is also possible to override the default memory types for individual functions and data objects using pragma directives or memory attribute keywords.

Guidelines for placing code

If your code fits in near memory, use the small code model. Otherwise, try the medium code model. If you have some functions that do not fit in a single 64-Kbyte section, you should prefer making those functions huge explicitly, rather than using the large code model which makes all functions huge by default.

Guidelines for placing data

If all your variables fit in tiny memory, you should use the small data model. If they almost fit, it is probably better to explicitly make some large, infrequently used variables near, instead of using the medium data model. If you use the medium or large data models, you can improve the efficiency of the code by explicitly making frequently used variables tiny.

Direct access to tiny variables is more efficient than direct access to near variables. However, the natural pointer size for the STM8 architecture is 16 bits. Therefore, to generate the most efficient code for pointer access, you should prefer near pointers. This is the default in both the small and the medium data models. You should only use tiny pointers to save space in large data structures, at the expense of larger and slower code.

Because there is no ROM in tiny memory, constants are placed in near memory in both the small and the medium data model. (If you explicitly make a constant tiny, it will be allocated in RAM and initialized by copy during startup.) If the constants do not fit in near memory, some constants must be made far. You should prefer to do this explicitly, rather than using the large data model, because the default pointer is far in this data model. Far pointers only have limited support in the STM8 architecture. The large data model should therefore only be used if you need to pass pointers to far constants as arguments to standard library functions.

Huge variables and pointers should only be used if you have constant objects that are too large to fit in a single 64-Kbyte section. This should be a very rare situation, so there is no data model that makes constants or pointers huge by default.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.
- Use `__tiny` pointers to save data memory at the expense of code memory and execution speed.

FUNCTION PROTOTYPES

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

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. Using the prototyped style will also make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

To make the compiler verify that all functions have proper prototypes, use the compiler option **Require prototypes** (`--require_prototypes`).

Prototyped style

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

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

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

Kernighan & Ritchie style

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

For example:

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

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

INTEGER TYPES AND BIT NEGATION

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

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

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

Here, the test is always false. On the right hand side, `0x80` is `0x0080`, and `~0x0080` becomes `0xFF7F`. On the left hand side, `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 details, see *__monitor*, page 244.

To read more about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 234.

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 STM8 devices are included in the IAR product installation. The header files are named `io`*`device`*`.h` and define the processor-specific special function registers (SFRs).

Note: Each header file contains one section used by the compiler, and one section used by the assembler.

SFRs with bitfields are declared in the header file. This example is from `iostm8s208mb.h`:

```
/* Flash control register 1 */
__no_init volatile union
{
  unsigned char FLASH_CR1;
  struct
  {
    unsigned char FIX         : 1;
    unsigned char IE          : 1;
    unsigned char AHALT       : 1;
    unsigned char HALT        : 1;
  } FLASH_CR1_bit;
} @ 0x505A;

/* 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 */
  FLASH_CR1 = 0x03;

  /* Bitfield accesses */
  FLASH_CR1_bit.FIX  = 1;
  FLASH_CR1_bit.IE  = 1;
}
```

You can also use the header files as templates when you create new header files for other STM8 devices. For details about the `@` operator, see *Controlling data and function placement in memory*, page 150.

NON-INITIALIZED VARIABLES

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

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in a separate section, according to the specified memory keyword.

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

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

For information about the `__no_init` keyword, see page 246. Note that to use this keyword, language extensions must be enabled; see *-e*, page 190. For information about the `#pragma object_attribute`, see page 258.

Part 2. Reference information

This part of the IAR C/C++ Development Guide for STM8 contains these chapters:

- External interface details
- Compiler options
- Linker options
- Data representation
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- The linker configuration file
- Section reference
- IAR utilities
- Implementation-defined behavior.

External interface details

This chapter provides reference information about how the compiler and linker interact with their 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 and linker output.

Invocation syntax

You can use the compiler and linker either from the IDE or from the command line. Refer to the *IAR Embedded Workbench® IDE User Guide* for information about using the build tools from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccstm8 [options] [sourcefile] [options]
```

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

```
iccstm8 prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order that they are specified on the command line.

If you run the compiler from the command line without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

ILINK INVOCATION SYNTAX

The invocation syntax for ILINK is:

```
ilinkstm8 [arguments]
```

Each argument is either a command-line option, an object file, or a library.

For example, when linking the object file `prog.o`, use this command:

```
ilinkstm8 prog.o --config myConfigfile.icf
```

If no filename extension is specified for the linker configuration file, the configuration file must have the extension `icf`.

Generally, the order of arguments on the command line is *not* significant. There is, however, one exception: when you supply several libraries, the libraries are searched in the same order that they are specified on the command line. The default libraries are always searched last.

The output executable image will be placed in a file named `a.out`, unless the `-o` option is used.

If you run ILINK from the command line without any arguments, the ILINK version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

PASSING OPTIONS

There are three different ways of passing options to the compiler and to ILINK:

- Directly from the command line

  Specify the options on the command line after the `iccstm8` or `ilinkstm8` commands; see *Invocation syntax*, page 167.
- Via environment variables

  The compiler and linker automatically append the value of the environment variables to every command line; see *Environment variables*, page 169.
- Via a text file, using the `-f` option; see *-f*, page 191.

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 5.n\stm8\inc;c:\headers`
QCCSTM8	Specifies command line options; for example: `QCCSTM8=-lA asm.lst`

Table 26: Compiler environment variables

This environment variable can be used with ILINK:

Environment variable	Description
ILINKSTM8_CMD_LINE	Specifies command line options; for example: `ILINKSTM8_CMD_LINE=--config full.icf --silent`

Table 27: ILINK environment variables

Include file search procedure

This is a detailed description of the compiler's `#include` file search procedure:

- If the name of the `#include` file is an absolute path, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:

  ```
  #include <stdio.h>
  ```

  it searches these directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 242.

  2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 169.

  3 The automatically set up library system include directories. See *--dlib*, page 238 and *--dlib_config*, page 238.

- 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:

```
iccstm8 ..\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.

For information about the syntax for including header files, see *Overview of the preprocessor*, page 339.

Compiler output

The compiler can produce the following output:

- A linkable object file

  The object files produced by the compiler use the industry-standard format ELF. By default, the object file has the filename extension `o`.

- Optional list files

  Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 192. 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. To read more about diagnostic messages, see *Diagnostics*, page 172.

- 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 171.

- Size information

  Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as *shared*.

  Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

Error return codes

The compiler and linker return status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

Code	Description
0	Compilation or linking successful, but there might have been warnings.
1	Warnings were produced and the option `--warnings_affect_exit_code` was used.
2	Errors occurred.
3	Fatal errors occurred, making the tool abort.
4	Internal errors occurred, making the tool abort.

Table 28: Error return codes

ILINK output

ILINK can produce the following output:

- An absolute executable image

  The final output produced by the IAR ILINK Linker is an absolute object file containing the executable image that can be put into an EPROM, downloaded to a hardware emulator, or executed on your PC using the IAR C-SPY Debugger Simulator. By default, the file has the filename extension `out`. The output format is always in ELF, which optionally includes debug information in the DWARF format.
- Optional logging information

  During operation, ILINK logs its decisions on `stdout`, and optionally to a file. For example, if a library is searched, whether a required symbol is found in a library module, or whether a module will be part of the output. Timing information for each ILINK subsystem is also logged. For more information, see --*log*, page 216 and --*log_file*, page 216.
- Optional map files

  A linker map file—containing summaries of linkage, runtime attributes, memory, and placement, as well as an entry list— can be generated by the ILINK option `--map`, see --*map*, page 217. By default, the map file has the filename extension `map`.
- Diagnostic messages

  Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in the optional map file. To read more about diagnostic messages, see *Diagnostics*, page 172.
- Error return codes

  ILINK returns status information to the operating system which can be tested in a batch file, see *Error return codes*, page 171.
- Size information about used memory and amount of time

  Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` and displayed on the screen.

Diagnostics

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

MESSAGE FORMAT FOR THE COMPILER

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

filename,*linenumber* *level*[*tag*]: *message*

with these elements:

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.

MESSAGE FORMAT FOR THE LINKER

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

level[*tag*]: *message*

with these elements:

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 map file.

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler or linker finds a construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 202.

Warning

A diagnostic message that is produced when the compiler or linker finds a problem which is of concern, but not so severe as to prevent the completion of compilation or linking. Warnings can be disabled by use of the command line option `--no_warnings`, see page 198.

Error

A diagnostic message that is produced when the compiler or linker finds a serious error. An error will produce a non-zero exit code.

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of compiler options*, page 180, for a description of the compiler options that are available for setting severity levels.

For the compiler see also the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler or linker. It is produced using this form:

```
Internal error: message
```

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Include enough information to reproduce the problem, typically:

- The product name
- The version number of the compiler or of ILINK, which can be seen in the header of the list or map files generated by the compiler or by ILINK, respectively
- Your license number
- The exact internal error message text
- The files involved of the application that generated the internal error
- A list of the options that were used when the internal error occurred.

Compiler options

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

Options syntax

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

Refer to the *IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IDE and how to set them.

TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it can have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 168.

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 also take a path. The path can be relative or absolute. For example, to generate a listing to the file `List.lst` in the directory `..\listings\`:

  ```
  iccstm8 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:

  ```
  iccstm8 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:

  ```
  iccstm8 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:

  ```
  iccstm8 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`:

  ```
  iccstm8 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
--c89	Uses the C89 standard
--char_is_signed	Treats char as signed
--char_is_unsigned	Treats char as unsigned
--code_model	Specifies the code model
--core	Specifies a CPU core
-D	Defines preprocessor symbols
--data_model	Specifies the data model
--debug	Generates debug information
--dependencies	Lists file dependencies
--diag_error	Treats these as errors
--diag_remark	Treats these as remarks
--diag_suppress	Suppresses these diagnostics
--diag_warning	Treats these as warnings
--diagnostics_tables	Lists all diagnostic messages
--discard_unused_publics	Discards unused public symbols
--dlib_config	Uses the system header files for the DLIB library and determines which configuration of the library to use
-e	Enables language extensions
--ec++	Enables Embedded C++ syntax
--eec++	Enables Extended Embedded C++ syntax
--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
--header_context	Lists all referred source files and header files
-I	Specifies include file path
-l	Creates a list file
--mfc	Enables multi-file compilation

Table 29: Compiler options summary

Command line option	Description
`--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`	Enables verbose logging of MISRA C checking. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--no_code_motion`	Disables code motion optimization
`--no_cross_call`	Disables cross-call optimization
`--no_cse`	Disables common subexpression elimination
`--no_fragments`	Disables section fragment handling
`--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_static_destruction`	Disables destruction of C++ static variables at program exit
`--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`.
`--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

Table 29: Compiler options summary (Continued)

Command line option	Description
`--public_equ`	Defines a global named assembler label
`-r`	Generates debug information. Alias for `--debug`.
`--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_unix_directory_separators`	Uses / as directory separator in paths
`--vla`	Enables VLA support
`--vregs`	Specifies the number of virtual byte registers
`--warnings_affect_exit_code`	Warnings affects exit code
`--warnings_are_errors`	Warnings are treated as errors

Table 29: Compiler options summary (Continued)

Descriptions of options

The following section gives detailed reference information about each compiler option.

Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--c89

Syntax

`--c89`

Description

Use this option to enable the C89 standard instead of Standard C.

Note: This option is mandatory when the MISRA C checking is enabled.

See also

C language overview, page 117.

Project>Options>General Options>C/C++ Compiler>Language>C89

--char_is_signed

Syntax `--char_is_signed`

Description By default, the compiler interprets the plain `char` type as unsigned. Use this option to make the compiler interpret the plain `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

Note: The runtime library is compiled without the `--char_is_signed` option and cannot be used with code that is compiled with this option.

Project>Options>C/C++ Compiler>Language>Plain 'char' is

--char_is_unsigned

Syntax `--char_is_unsigned`

Description Use this option to make the compiler interpret the plain `char` type as unsigned. This is the default interpretation of the plain `char` type.

Project>Options>C/C++ Compiler>Language>Plain 'char' is

--code_model

Syntax `--code_model={small|medium|large}`

Parameters

`small`	Uses the small code model
`medium` (default)	Uses the medium code model
`large`	Uses the large code model

Description Use this option to select the code model. If you do not select a code model, the compiler uses the default code model. Note that all modules of your application must use the same code model.

See also *Code models and memory attributes for function storage*, page 35 and *Efficient use of memory types*, page 159.

Project>Options>General Options>Target>Code model

--core

Syntax `--core=stm8`

Parameters

`stm8`	Generates code for the STM8 instruction set architecture

Description

Use this option to select the processor core for which the code will be generated. If you do not use the option to specify a core, the compiler uses the default core.

Note: The only available core is `stm8`.

Project>Options>General Options>Target>Core

-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:

`-D`*`symbol`*

is equivalent to:

`#define` *`symbol`* `1`

To get the equivalence of:

`#define FOO`

specify the `=` sign but nothing after, for example:

`-DFOO=`

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--data_model

Syntax
`--data_model={small|medium|large}`

Parameters

`small`	Uses the small data model
`medium` (default)	Uses the medium data model
`large`	Uses the large data model

Description
Use this option to select the data model. 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 24 and *Efficient use of memory types*, page 159.

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 178.

Description

Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension `i`.

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened input file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output is in makefile style. For each input file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of an input file. For example:

```
foo.o: c:\iar\product\include\stdio.h
foo.o: d:\myproject\include\foo.h
```

An example of using `--dependencies` with a popular make utility, such as gmake (GNU make):

1 Set up the rule for compiling files to be something like:

```
%.o : %.c
      $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

That is, in addition to producing an object file, the command also produces a dependency file in makefile style (in this example, using the extension `.d`).

2 Include all the dependency files in the makefile using, for example:

```
-include $(sources:.c=.d)
```

Because of the dash (`-`) it works the first time, when the `.d` files do not yet exist.

This option is not available in the IDE.

--diag_error

Syntax

`--diag_error=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be

generated. The exit code will be non-zero. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors

--diag_remark

Syntax

`--diag_remark=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`

Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks

--diag_suppress

Syntax

`--diag_suppress=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax `--diag_warning=`*`tag`*`[,`*`tag`*`,...]`

Parameters

`tag`	The number of a diagnostic message, for example the message number `Pe826`

Description Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax `--diagnostics_tables {`*`filename`*`|`*`directory`*`}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 178.

Description Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

This option is not available in the IDE.

--discard_unused_publics

Syntax `--discard_unused_publics`

Description Use this option to discard unused public functions and variables when compiling with the `--mfc` compiler option.

Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output.

See also

--mfc, page 193 and *Multi-file compilation units*, page 154.

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 178.
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

Each runtime library has a corresponding library configuration file. Use this option to explicitly specify which library configuration file to use, either by specifying an explicit file or by specifying 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 `stm8\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 67.

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 75.

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.

Project>Options>C/C++ Compiler>Language>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>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 128.

Project>Options>C/C++ Compiler>Language>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>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 178.

Descriptions

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**.

--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.

Note that, for this option, the directories are searched in the same order that they are specified on the command line.

See also *Include file search procedure*, page 169.

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 178.

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

--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

```
iccstm8 myfile1.c myfile2.c myfile3.c --mfc
```

See also

--discard_unused_publics, page 188, *--output, -o*, page 199, and *Multi-file compilation units*, page 154.

Project>Options>C/C++ Compiler>Multi-file compilation

--no_code_motion

Syntax

```
--no_code_motion
```

Description

Use this option to disable code motion optimizations. These optimizations, which are performed at the optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code might be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cross_call

Syntax

```
--no_cross_call
```

Description

Use this option to disable the cross-call optimization. This optimization is performed at size optimization, level High. Note that, although the option can drastically reduce the code size, this option increases the execution time.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Cross call

--no_cse

Syntax

```
--no_cse
```

Description

Use this option to disable common subexpression elimination. At the optimization levels Medium and High, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_fragments

Syntax
```
--no_fragments
```

Description

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image. The effect of using this option in the compiler is smaller object size, but a larger executable image.

See also

Keeping symbols and sections, page 57.

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. Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level **High**, normally reduces execution time, but the resulting code might be difficult to debug.

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

If you do not want to disable inlining for a whole module, use `#pragma inline=never` on an individual function instead.

Note: This option has no effect at optimization levels below **High**.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_path_in_file_macros

Syntax
```
--no_path_in_file_macros
```

Description

Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

See also *Descriptions of predefined preprocessor symbols*, page 270.

This option is not available in the IDE.

--no_static_destruction

Syntax `--no_static_destruction`

Description Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.

Use this option to suppress the emission of such code.

See also *Setting up the atexit limit*, page 87.

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 189, and *--system_include_dir*, page 203.

Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories

--no_tbaa

Syntax `--no_tbaa`

Description Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.

See also *Type-based alias analysis*, page 157.

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. The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities.

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

Note: This option has no effect at optimization levels below High.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax `--no_warnings`

Description By default, the compiler issues warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax `--no_wrap_diagnostics`

Description By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

-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

`n`	None* (Best debug support)
`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 153.

Project>Options>C/C++ Compiler>Optimizations

--only_stdout

Syntax

`--only_stdout`

Description

Use this option to make the compiler use the standard output stream (`stdout`) 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 178.

Description

By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `o`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--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 178.

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 178.

Description Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

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 178.

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.

--relaxed_fp

Syntax

```
--relaxed_fp
```

Description

Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:

- The expression consists of both single- and double-precision values
- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

Note that performing the calculation in single precision instead of double precision might cause a loss of accuracy.

Example

```
float f(float a, float b)
{
  return a + b * 3.0;
}
```

The C standard states that `3.0` in this example has the type `double` and therefore the whole expression should be evaluated in `double` precision. However, when the

--relaxed_fp option is used, 3.0 will be converted to float and the whole expression can be evaluated in float precision.

Project>Options>General Options>Language>Relaxed floating-point precision

--remarks

Syntax `--remarks`

Description The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also *Severity levels*, page 174.

Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax `--require_prototypes`

Description Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

Project>Options>C/C++ Compiler>Language>Require prototypes

--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.

Project>Options>C/C++ Compiler>Language>Language conformances>Strict standard C

--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 178.

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 189, and *--no_system_include*, page 196.

This option is not available in the IDE.

--use_unix_directory_separators

Syntax

`--use_unix_directory_separators`

Description

Use this option to make DWARF debug information use `/` (instead of `\`) as directory separators in file paths.

This option can be useful if you have a debugger that requires directory separators in UNIX style.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--vla

Syntax `--vla`

Description Use this option to enable variable length arrays. Note that this option requires Standard C and cannot be used when the `--c89` compiler option is used.

See also *C language overview*, page 117.

Project>Options>General Options>C/C++ Compiler>Language>Allow VLA

--vregs

Syntax `--vregs={12|16}`

Parameters

`12`	Makes the compiler use 12 virtual byte registers.
`16` (default)	Makes the compiler use 16 virtual byte registers.

Description Use this option to specify the number of virtual byte registers that are available to the compiler.

See also *Virtual registers*, page 102.

To set this option, use **Project>Options>C/C++ Compiler>Optimizations>Number of virtual byte registers**

--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 188.

Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors

Linker options

This chapter gives detailed reference information about each linker option.

For general syntax rules, see *Options syntax*, page 177.

Summary of linker options

This table summarizes the linker options:

Command line option	Description
`--config`	Specifies the linker configuration file to be used by the linker
`--config_def`	Defines symbols for the configuration file
`--cpp_init_routine`	Specifies a user-defined C++ dynamic initialization routine
`--debug_lib`	Uses the C-SPY debug library
`--define_symbol`	Defines symbols that can be used by the application
`--dependencies`	Lists file dependencies
`--diag_error`	Treats these message tags as errors
`--diag_remark`	Treats these message tags as remarks
`--diag_suppress`	Suppresses these diagnostic messages
`--diag_warning`	Treats these message tags as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--entry`	Treats the symbol as a root symbol and as the start of the application
`--error_limit`	Specifies the allowed number of errors before linking stops
`--export_builtin_config`	Produces an `icf` file for the default configuration
`-f`	Extends the command line
`--force_output`	Produces an output file even if errors occurred
`--image_input`	Puts an image file in a section
`--keep`	Forces a symbol to be included in the application
`--log`	Enables log output for selected topics
`--log_file`	Directs the log to a file

Table 30: Linker options summary

Command line option	Description
--mangled_names_in_messages	Adds mangled names in messages
--map	Produces a map file
--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	Enables verbose logging of MISRA C checking. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* and the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
--no_fragments	Disables section fragment handling
--no_library_search	Disables automatic runtime library search
--no_locals	Removes local symbols from the ELF executable image.
--no_range_reservations	Disables range reservations for absolute symbols
--no_remove	Disables removal of unused sections
--no_warnings	Disables generation of warnings
--no_wrap_diagnostics	Does not wrap long lines in diagnostic messages
-o	Sets the object filename, alias for --output
--only_stdout	Uses standard output only
--output	Sets the object filename
--place_holder	Reserve a place in ROM to be filled by some other tool, for example a checksum calculated by ielftool.
--redirect	Redirects a reference to a symbol to another symbol
--remarks	Enables remarks
--silent	Sets silent operation
--strip	Removes debug information from the executable image
--warnings_are_errors	Warnings are treated as errors
--warnings_affect_exit_code	Warnings affects exit code

Table 30: Linker options summary (Continued)

Descriptions of options

The following section gives detailed reference information about each compiler and linker 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.

--config

Syntax `--config *filename*`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.

Description Use this option to specify the configuration file to be used by the linker (the default filename extension is `icf`). If no configuration file is specified, a default configuration is used. This option can only be used once on the command line.

See also The chapter *The linker configuration file*.

Project>Options>Linker>Config>Linker configuration file

--config_def

Syntax `--config_def *symbol*[=*constant_value*]`

Parameters

symbol	The name of the symbol to be used in the configuration file. By default, the value 0 (zero) is used.
constant_value	The constant value of the configuration symbol.

Description Use this option to define a constant configuration symbol to be used in the configuration file. This option has the same effect as the `define symbol` directive in the linker configuration file. This option can be used more that once on the command line.

See also *--define_symbol*, page 210 and *Interaction between ILINK and the application*, page 61.

Project>Options>Linker>Config>Defined symbols for configuration file

--cpp_init_routine

Syntax `--cpp_init_routine` *`routine`*

Parameters

`routine`	A user-defined C++ dynamic initialization routine.

Description When using the IAR C/C++ compiler and the standard library, C++ dynamic initialization is handled automatically. In other cases you might need to use this option.

If any sections with the section type `INIT_ARRAY` or `PREINIT_ARRAY` are included in your application, the C++ dynamic initialization routine is considered to be needed. By default, this routine is named `__iar_cstart_call_ctors` and is called by the startup code in the standard library. Use this option if you are not using the standard library and require another routine to handle these section types.

To set this option, use **Project>Options>Linker>Extra Options**.

--debug_lib

Syntax `--debug_lib`

Description Use this option to include the C-SPY debug library.

See also *Application debug support*, page 71 for more information about the C-SPY debug library.

Project>Options>Linker>Library>Include C-SPY debugging support

--define_symbol

Syntax `--define_symbol` *`symbol=constant_value`*

Parameters

`symbol`	The name of the constant symbol that can be used by the application.
`constant_value`	The constant value of the symbol.

Description Use this option to define a constant symbol, that is a label, that can be used by your application. If no value is specified, `0` is used. This option can be used more than once

on the command line. Note that his option is different from the `define symbol` directive.

See also

--config_def, page 209 and *Interaction between ILINK and the application*, page 61.

Project>Options>Linker>#define>Defined symbols

--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 178.

Description

Use this option to make the linker list the names of the linker configuration, object, and library files opened for input into a file with the default filename extension `i`.

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened input file, including the full path, if available, is output on a separate line. For example:

```
c:\myproject\foo.o
d:\myproject\bar.o
```

If `--dependencies=m` is used, the output is in makefile style. For each input file, one line containing a makefile dependency rule is produced. Each line consists of the name of the output file, a colon, a space, and the name of an input file. For example:

```
a.out: c:\myproject\foo.o
a.out: d:\myproject\bar.o
```

This option is not available in the IDE.

--diag_error

Syntax

```
--diag_error=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to reclassify certain diagnostic messages as errors. An error indicates a problem of such severity that an executable image will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.

Project>Options>Linker>Diagnostics>Treat these as errors

--diag_remark

Syntax

`--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 construction that may cause strange behavior in the executable image. 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>Linker>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>Linker>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 linker to stop before linking is completed. This option may be used more than once on the command line.

Project>Options>Linker>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 178.

Description

Use this option to list all possible diagnostic messages in a named file.

This option cannot be given together with other options.

This option is not available in the IDE.

--entry

Syntax `--entry `*`symbol`*

Parameters

symbol	The name of the symbol to be treated as a root symbol and start label

Description

Use this option to make a symbol be treated as a root symbol and the start label of the application. This is useful for loaders. If this option is not used, the default start symbol is `__iar_program_start`. A root symbol is kept whether or not it is referenced from

the rest of the application, provided its module is included. A module in an object file is always included and a module part of a library is only included if needed.

Project>Options>Linker>Library>Override default program entry

--error_limit

Syntax `--error_limit=n`

Parameters

n	The number of errors before the linker stops linking. *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 linker stops the linking. By default, 100 errors are allowed.

This option is not available in the IDE.

--export_builtin_config

Syntax `--export_builtin_config filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.

Description Exports the configuration used by default to a file.

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 178.

Descriptions

Use this option to make the linker read command line options from the named file, with the default filename extension `xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

To set this option, use **Project>Options>Linker>Extra Options**.

--force_output

Syntax

`--force_output`

Description

Use this option to produce an output executable image regardless of any linking errors.

To set this option, use **Project>Options>Linker>Extra Options**

--image_input

Syntax

`--image_input filename [symbol,[section[,alignment]]]`

Parameters

filename	The pure binary file containing the raw image you want to link
symbol	The symbol which the binary data can be referenced with.
section	The section where the binary data will be placed; default is `.text`.
alignment	The alignment of the section; default is `1`.

Description

Use this option to link pure binary files in addition to the ordinary input files. The file's entire contents are placed in the section, which means it can only contain pure binary data.

The section where the contents of the *filename* file are placed, is only included if the symbol *symbol* is required by your application. Use the `--keep` option if you want to force a reference to the section.

Example

```
--image_input bootstrap.abs,Bootstrap,CSTARTUPCODE,4
```

The contents of the pure binary file `bootstrap.abs` are placed in the section `CSTARTUPCODE`. The section where the contents are placed is 4-byte aligned and will

only be included if your application (or the command line option `--keep`) includes a reference to the symbol `Bootstrap`.

See also

--*keep*, page 216.

Project>Options>Linker>Input>Raw binary image

--keep

Syntax

`--keep symbol`

Parameters

symbol	The name of the symbol to be treated as a root symbol

Description

Normally, the linker keeps a symbol only if it is needed by your application. Use this option to make a symbol always be included in the final application.

Project>Options>Linker>Input>Keep symbols

--log

Syntax

`--log topic,topic,...`

Parameters

`initialization`	Log initialization decisions
`modules`	Log module selections
`sections`	Log section selections

Description

Use this option to make the linker log information to `stdout`. The log information can be useful for understanding why an executable image became the way it is.

See also

--*log_file*, page 216.

Project>Options>Linker>List>Generate log

--log_file

Syntax

`--log_file filename`

Parameters — For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.

Description — Use this option to direct the log output to the specified file.

See also — *--log*, page 216.

Project>Options>Linker>List>Generate log

--mangled_names_in_messages

Syntax — `--mangled_names_in_messages`

Descriptions — Use this option to produce both mangled and unmangled names for C/C++ symbols in messages. Mangling is a technique used for mapping a complex C name or a C++ name (for example, for overloading) into a simple name. For example, `void h(int, char)` becomes `_Z1hic`.

This option is not available in the IDE.

--map

Syntax — `--map {filename|directory}`

Description — Use this option to produce a linker memory map file. The map file has the default filename extension `map`. The map file contains:

- Linking summary in the map file header which lists the version of the linker, the current date and time, and the command line that was used.
- Runtime attribute summary which lists IAR-specific runtime attributes.
- Placement summary which lists each section/block in address order, sorted by placement directives.
- Initialization table layout which lists the data ranges, packing methods, and compression ratios.
- Module summary which lists contributions from each module to the image, sorted by directory and library.
- Entry list which lists all public and some local symbols in alphabetical order, indicating which module they came from.
- 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.

This option can only be used once on the command line.

Project>Options>Linker>List>Generate linker map file

--no_fragments

Syntax

```
--no_fragments
```

Description

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image.

See also

Keeping symbols and sections, page 57.

To set this option, use **Project>Options>Linker>Extra Options**

--no_library_search

Syntax

```
--no_library_search
```

Description

Use this option to disable the automatic runtime library search. This option turns off the automatic inclusion of the correct standard libraries. This is useful, for example, if the application needs a user-built standard library, etc.

Project>Options>Linker>Library>Automatic runtime library selection

--no_locals

Syntax `--no_locals`

Description Use this option to remove local symbols from the ELF executable image.

Note: This option does not remove any local symbols from the DWARF information in the executable image.

Project>Options>Linker>Output

--no_range_reservations

Syntax `--no_range_reservations`

Description Normally, the linker reserves any ranges used by absolute symbols with a non-zero size, excluding them from consideration for `place in` commands.

When this option is used, these reservations are disabled, and the linker is free to place sections in such a way as to overlap the extent of absolute symbols.

To set this option, use **Project>Options>Linker>Extra Options**

--no_remove

Syntax `--no_remove`

Description When this option is used, unused sections are not removed. In other words, each module that is included in the executable image contains all its original sections.

See also *Keeping symbols and sections*, page 57.

To set this option, use **Project>Options>Linker>Extra Options**

--no_warnings

Syntax `--no_warnings`

Description By default, the linker issues warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax `--no_wrap_diagnostics`

Description By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

--only_stdout

Syntax `--only_stdout`

Description Use this option to make the linker use the standard output stream (`stdout`) 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 178.

Description By default, the object executable image produced by the linker is located in a file with the name `a.out`. Use this option to explicitly specify a different output filename, which by default will have the filename extension `out`.

Project>Options>Linker>Output>Output file

--place_holder

Syntax `--place_holder symbol[,size[,section[,alignment]]]`

Parameters

symbol	The name of the symbol to create
size	Size in ROM; by default 4 bytes
section	Section name to use; by default `.text`

alignment	Alignment of section; by default `1`

Description

Use this option to reserve a place in ROM to be filled by some other tool, for example a checksum calculated by `ielftool`. Each use of this linker option results in a section with the specified name, size, and alignment. The symbol can be used by your application to refer to the section.

Note: Like any other section, sections created by the `--place_holder` option will only be included in your application if the section appears to be needed. The `--keep` linker option, or the `keep` linker directive can be used for forcing such section to be included.

See also

IAR utilities, page 315.

To set this option, use **Project>Options>Linker>Extra Options**

--redirect

Syntax

`--redirect` *`from_symbol=to_symbol`*

Parameters

from_symbol	The name of the source symbol
to_symbol	The name of the destination symbol

Description

Use this option to change a reference from one symbol to another symbol.

To set this option, use **Project>Options>Linker>Extra Options**

--remarks

Syntax

`--remarks`

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the linker does not generate remarks. Use this option to make the linker generate remarks.

See also

Severity levels, page 174.

Project>Options>Linker>Diagnostics>Enable remarks

--silent

Syntax `--silent`

Description

By default, the linker issues introductory messages and a final statistics report. Use this option to make the linker operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.

This option is not available in the IDE.

--strip

Syntax `--strip`

Description

By default, the linker retains the debug information from the input object files in the output executable image. Use this option to remove that information.

To set related options, choose:

Project>Options>Linker>Output>Include debug information in output

--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 linker treat all warnings as errors. If the linker encounters an error, no executable image is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the option `--diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

See also

--*diag_warning*, page 213 and --*diag_warning*, page 188.

Project>Options>Linker>Diagnostics>Treat all warnings as errors

Data representation

This chapter describes the data types, pointers, and structure types supported by the compiler.

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will have the same alignment as the structure member with the most strict alignment.

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.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

ALIGNMENT ON THE STM8 MICROCONTROLLER

There are no alignment requirements for how the STM8 microcontroller accesses memory.

Byte order

The STM8 microcontroller stores data in big-endian byte order.

In the little-endian byte order, the *least* significant byte is stored at the lowest address in memory. The *most* significant byte is stored at the highest address.

In the big-endian byte order, the *most* significant byte is stored at the lowest address in memory. The *least* significant byte is stored at the highest address. It might be necessary to use the `#pragma bitfields=reversed` directive to be compatible with code for other compilers and I/O register definitions of some devices, see *Bitfields*, page 227.

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	1
unsigned short	16 bits	0 to 65535	1
signed int	16 bits	-32768 to 32767	1
unsigned int	16 bits	0 to 65535	1
signed long	32 bits	-2^{31} to $2^{31}-1$	1
unsigned long	32 bits	0 to $2^{32}-1$	1
signed long long	32 bits	-2^{31} to $2^{31}-1$	1
unsigned long long	32 bits	0 to $2^{32}-1$	1

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 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 STM8, plain integer types are treated as unsigned.

Bitfields in expressions are treated as int if int can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.

Each bitfield is placed in the next container of its base type that has enough available bits to accommodate the bitfield. Within each container, the bitfield is placed in the first available byte or bytes, taking the byte order into account.

In addition, the compiler supports an alternative bitfield allocation strategy (disjoint types), where bitfield containers of different types are not allowed to overlap. Using this allocation strategy, each bitfield is placed in a new container if its type is different from that of the previous bitfield, or if the bitfield does not fit in the same container as the previous bitfield. Within each container, the bitfield is placed from the least significant bit to the most significant bit (disjoint types) or from the most significant bit to the least significant bit (reverse disjoint types). This allocation strategy will never use less space than the default allocation strategy (joined types), and can use significantly more space when mixing bitfield types.

Use the `#pragma bitfield` directive to choose which bitfield allocation strategy to use, see *bitfields*, page 251.

Example

Assume this example:

```
struct bitfield_example
{
  uint32_t a : 12;
  uint16_t b : 3;
  uint16_t c : 7;
  uint8_t  d;
};
```

The example in the joined types bitfield allocation strategy

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset `0` and puts `a` into the first and second bytes of the container.

For the second bitfield, `b`, a 16-bit container is needed and because there are still four bits free at offset `0`, the bitfield is placed there.

For the third bitfield, `c`, as there is now only one bit left in the first 16-bit container, a new container is allocated at offset `2`, and `c` is placed in the first byte of this container.

The fourth member, `d`, can be placed in the next available full byte, which is the byte at offset `3`.

In each case, each bitfield is allocated starting from the most significant free bit of its container to ensure that it is placed into bytes from left to right.

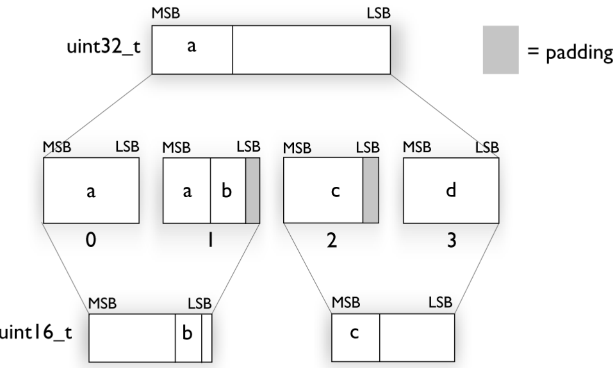


Figure 15: Layout of bitfield members for joined types in big-endian mode

The example in the disjoint types bitfield allocation strategy

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset `0` and puts `a` into the least significant 12 bits of the container.

To place the second bitfield, `b`, a new container is allocated at offset `4`, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

The fourth member, `d`, is allocated into the byte at offset `6`. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order (reverse disjoint types), each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example`:

Figure 16: Layout of bitfield_example for disjoint types

FLOATING-POINT TYPES

In the IAR C/C++ Compiler for STM8, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

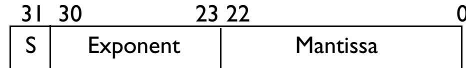
Type	Size
float	32 bits
double	32 bits
long double	32 bits

Table 32: Floating-point types

Exception flags according to the IEEE 754 standard are not supported.

32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:



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:

`±1.18E-38 to ±3.39E+38`

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 7 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 denormalized, even though the number is treated as if the exponent was 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a denormalized number is:

  ```
  (-1)^S * 2^(1-BIAS) * 0.Mantissa
  ```

  where `BIAS` is 127 for 32-bit floating-point values.

Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

These function pointers are available:

Keyword	Address range	Pointer size	Description
`__near_func`	`0x0-0xFFFF`	16 bits	Can only point to `__near_func` functions.
`__far_func`	`0x0-0xFFFFFF`	24 bits	Can point to both `__far_func` and `__huge_func` functions.

Table 33: Function pointers

Keyword	Address range	Pointer size	Description
`__huge_func`	`0x0-0xFFFFFF`	24 bits	Can point to both `__far_func` and `__huge_func` functions.

Table 33: Function pointers (Continued)

DATA POINTERS

These data pointers are available:

Keyword	Pointer size	Index type	Pointer value range	Address range
`__tiny`	8 bits	signed short	signed char	`0x0-0xFF`
`__near`	16 bits	signed int	signed short	`0x0-0xFFFF`
`__far`	24 bits	signed int	signed short	`0x0-0xFFFFFF` (16-bit arithmetics)
`__huge`	24 bits	signed long	signed long	`0x0-0xFFFFFF`
`__eeprom`	16 bits	signed int	signed short	`0x0-0xFFFF`

Table 34: Data pointers

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension
- Casting from a larger pointer to a smaller pointer is performed by truncation
- Casting `__eeprom` to another data pointer and vice versa is illegal

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 STM8, the size of `size_t` is 16 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 STM8, the size of ptrdiff_t is 16 bits.

Note: Subtracting the start address of an object from the end address can yield a negative value, because the object can be larger than what the ptrdiff_t can represent. See this example:

```
char buff[60000];           /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff;            /* signed integer type. */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;
```

intptr_t

intptr_t is a signed integer type large enough to contain a void *. In the IAR C/C++ Compiler for STM8, the size of intptr_t is 16 bits in the small and medium data models, and 32 bits in the large data model.

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.

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:

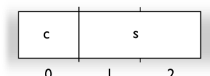


Figure 17: Structure layout

The alignment of the structure is 1 byte, and the size is 3 bytes.

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 STM8 are described below.

Rules for accesses

In the IAR C/C++ Compiler for STM8, 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 8-bit data types for all memory types.

For all combinations of object types not listed, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, the `volatile` object will be write-protected, but nothing else will change. This can be used for protecting objects stored in flash memory.

To protect an object in flash memory from write accesses, define the variables like this:

```
const volatile int x @ "FLASH";
```

The compiler will generate the read write section `FLASH`. That section should be placed in ROM and is used for manually initializing the variables when the application starts up.

Thereafter, the initializers can be reflashed with other values at any time.

DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` and located in the memories `__near`, `__far`, and `__huge` are allocated in ROM. For `__tiny`, 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 STM8 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 *Section 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 STM8 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 detailed information about each attribute, see *Descriptions of extended keywords*, page 242.

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 190 for additional information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

- Available *function memory attributes*: `__near_func`, `__far_func`, and `__huge_func`
- Available *data memory attributes*: `__tiny`, `__near`, `__far`, `__huge` and `__eeprom`.

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is used. You can specify one memory attribute for each level of pointer indirection.

General type attributes

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt` and `__task`
- *Data type attributes*: `const` and `volatile`.

You can specify as many type attributes as required for each level of pointer indirection.

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 234.

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 `__near` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in near memory. The variables `k` and `l` behave in the same way:

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

Note that the attribute affects both identifiers.

This declaration of i and j is equivalent with the previous one:

```
#pragma type_attribute=__near
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 30.

An easier way of specifying storage is to use type definitions. These two declarations are equivalent:

```
typedef char __near Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__near char b;
char __near *bp;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

`int __near * p;`	The `int` object is located in `__near` memory.
`int * __near p;`	The pointer is located in `__near` memory.
`__near int * p;`	The pointer is located in `__near` memory.

Syntax for type attributes on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```
__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);
```

Syntax for type attributes on function pointers

To declare a function pointer, use this syntax:

```
int (__far_func * fp) (double);
```

After this declaration, the function pointer fp points to far memory.

An easier way of specifying storage is to use type definitions:

```
typedef __far_func void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

Note that #pragma type_attribute can be used together with a typedef declaration.

OBJECT ATTRIBUTES

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 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, @, __root, __task, and __weak
- Object attributes that can be used for functions: __intrinsic, __monitor, and __noreturn.

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 150.

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
__eeprom	Controls the storage of data objects
__far	Controls the storage of data objects
__far_func	Controls the storage of functions
__huge	Controls the storage of data objects
__huge_func	Controls the storage of functions
__interrupt	Supports interrupt functions
__intrinsic	Reserved for compiler internal use only
__monitor	Supports atomic execution of a function
__near	Controls the storage of data objects
__near_func	Controls the storage of functions
__no_init	Supports 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
__task	Relaxes the rules for preserving registers
__tiny	Controls the storage of data objects
__weak	Declares a symbol to be externally weakly linked

Table 35: Extended keywords summary

Descriptions of extended keywords

These sections give detailed information about each extended keyword.

__eeprom

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__eeprom` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in EEPROM memory. You can also use the `__eeprom` attribute to create a pointer explicitly pointing to an object located in the EEPROM memory.

Storage information

- Address range: `0-0xFFFF` (64 Kbytes)
- Maximum object size: 64 Kbytes-1
- Pointer size: 2 bytes.

Example

```
__eeprom int x;
```

See also

Memory types, page 26.

__far

Syntax

Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__far` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in far memory. You can also use the `__far` attribute to create a pointer explicitly pointing to an object located in the far memory.

Storage information

- Address range: `0-0xFFFFFF` (16 Mbytes)
- Maximum object size: 64 Kbytes-1. An object cannot cross a 64-Kbyte boundary.
- Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

Example

```
__far int x;
```

See also

Memory types, page 26.

__far_func

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__far_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in far memory. You can also use the `__far_func` attribute to create a pointer explicitly pointing to a function located in far memory.

Storage information

- Address range: `0-0xFFFFFF` (16 Mbytes)
- Maximum size: 64 Kbytes. A function cannot cross a 64-Kbyte boundary.
- Pointer size: 3 bytes

Example

```
__far_func void myfunction(void);
```

See also

Code models and memory attributes for function storage, page 35.

__huge

Syntax

Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 237.

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. You can also use the `__huge` attribute to create a pointer explicitly pointing to an object located in the huge memory.

Storage information

- Address range: `0-0xFFFFFF` (16 Mbytes)
- Maximum object size: 16 Mbytes-1
- Pointer size: 3 bytes. Arithmetics and comparison is performed on the entire 24-bit address.

Example

```
__huge int x;
```

See also

Memory types, page 26.

__huge_func

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__huge_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in huge memory. You can also use the `__huge_func` attribute to create a pointer explicitly pointing to a function located in huge memory.

Storage information

- Address range: `0-0xFFFFFF` (16 Mbytes)
- Maximum size: 16 Mbytes
- Pointer size: 3 bytes

Example

```
__huge_func void myfunction(void);
```

See also

Code models and memory attributes for function storage, page 35.

__interrupt

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

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=0x14
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 37, *.intvec*, page 311.

__intrinsic

Description

The `__intrinsic` keyword is reserved for compiler internal use only.

__monitor

Syntax

Follows the generic syntax rules for object attributes that can be used on functions, see *Object attributes*, page 240.

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 38. Read also about the intrinsic functions *__disable_interrupt*, page 266, *__enable_interrupt*, page 266, *__get_interrupt_state*, page 266, and *__set_interrupt_state*, page 267.

__near

Syntax

Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 237.

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: `0-0xFFFF` (64 Kbytes)
- Maximum object size: 65535 bytes
- Pointer size: 2 bytes.

Example

```
__near int x;
```

See also

Memory types, page 26.

__near_func

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

Description

The `__near_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in near memory. You can also use the `__near_func` attribute to create a pointer explicitly pointing to a function located in near memory.

Storage information

- Address range: `0-0xFFFF` (64 Kbytes)
- Maximum size: 64 Kbytes

- Pointer size: 2 bytes

Example

```
__near_func void myfunction(void);
```

See also *Code models and memory attributes for function storage*, page 35.

__no_init

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 240.

Description Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Example

```
__no_init int myarray[10];
```

See also *Do not initialize directive*, page 294.

__noreturn

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 240.

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 240.

Description A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.

Example

```
__root int myarray[10];
```

See also To read more about root symbols and how they are kept, see *Keeping symbols and sections*, page 57.

__task

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 237.

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);
```

__tiny

Syntax

Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 237.

Description

The `__tiny` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in tiny memory. You can also use the `__tiny` attribute to create a pointer explicitly pointing to an object located in the tiny memory.

Storage information

- Address range: `0-0xFF` (256 bytes)
- Maximum object size: 255 bytes
- Pointer size: 1 byte.

Example

```
__tiny int x;
```

See also

Memory types, page 26.

__weak

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 240.

Description

Using the `__weak` object attribute on an external declaration of a symbol makes all references to that symbol in the module weak.

Using the `__weak` object attribute on a public definition of a symbol makes that definition a weak definition.

The linker will not include a module from a library solely to satisfy weak references to a symbol, nor will the lack of a definition for a weak reference result in an error. If no definition is included, the address of the object will be zero.

When linking, a symbol can have any number of weak definitions, and at most one non-weak definition. If the symbol is needed, and there is a non-weak definition, this definition will be used. If there is no non-weak definition, one of the weak definitions will be used.

Example

```
extern __weak int foo; /* A weak reference */

__weak void bar(void); /* A weak definition */
{
  /* Increment foo if it was included */
  if (&foo != 0)
    ++foo;
}
```

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
`basic_template_matching`	Makes a template function fully memory-attribute aware
`bitfields`	Controls the order of bitfield members
`data_alignment`	Gives a variable a higher (more strict) alignment
`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 sections
`message`	Prints a message

Table 36: Pragma directives summary

Pragma directive	Description
object_attribute	Changes the definition of a variable or a function
optimize	Specifies the type and level of an optimization
__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	Declares a section 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
vector	Specifies the vector of an interrupt or trap function
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable

Table 36: Pragma directives summary (Continued)

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

basic_template_matching

Syntax

```
#pragma basic_template_matching
```

Description

Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications described in *Templates and data memory attributes*, page 133.

Example

```
#pragma basic_template_matching
```

```
template<typename T> void fun(T *);

fun((int __near *) 0); /* Template parameter T becomes
                          int __near */
```

bitfields

Syntax

```
#pragma bitfields=disjoint_types|joined_types|
                  reversed_disjoint_types|reversed|default}
```

Parameters

`disjoint_types`	Bitfield members are placed from the least significant bit to the most significant bit in the container type. Storage containers of bitfields with different base types will not overlap.
`joined_types`	Bitfield members are placed depending on the byte order. Storage containers of bitfields will overlap other structure members. For more details, see *Bitfields*, page 227.
`reversed_disjoint_types`	Bitfield members are placed from the most significant bit to the least significant bit in the container type. Storage containers of bitfields with different base types will not overlap.
`reversed`	This is an alias for `reversed_disjoint_types`.
`default`	Restores to default layout of bitfield members. The default behavior for the compiler is `joined_types`.

Description

Use this pragma directive to control the layout of bitfield members.

Example

```
#pragma bitfields=disjoint_types
/* Structure that uses disjoint bitfield types. */
{
  unsigned char error :1;
  unsigned char size  :4;
  unsigned short code  :10;
}
#pragma bitfields=default /* Restores to default setting. */
```

See also

Bitfields, page 227.

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.

diag_default

Syntax `#pragma diag_default=`*`tag`*`[,`*`tag`*`,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also *Diagnostics*, page 172.

diag_error

Syntax `#pragma diag_error=`*`tag`*`[,`*`tag`*`,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description | Use this pragma directive to change the severity level to `error` for the specified diagnostics.

See also | *Diagnostics*, page 172.

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 172.

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 172.

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 172.

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 169.

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 whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually occurs is subject to the compiler's heuristics.

`#pragma inline` is similar to the C++ keyword `inline`. The difference is that the compiler uses C++ inline semantics for the `#pragma inline` directive, but uses the Standard C semantics for the `inline` keyword.

Specifying `#pragma inline=never` disables the compiler's heuristics and makes sure that the function will not be inlined.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

Note: Because specifying `#pragma inline=forced` disables the compiler's heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels **None** or **Low**. No error or warning message will be emitted.

See also

Function inlining, page 156.

language

Syntax

```
#pragma language={extended|default|save|restore}
```

Parameters

`extended`	Enables the IAR Systems language extensions from the first use of the pragma directive and onward.
`default`	From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.
`save|restore`	Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code. Each use of `save` must be followed by a matching `restore` in the same file without any intervening `#include` directive.

Description

Use this pragma directive to control the use of language extensions.

Example 1

At the top of a file that needs to be compiled with IAR Systems extensions enabled:

```
#pragma language=extended
/* The rest of the file. */
```

Example 2

Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled:

```
#pragma language=extended
/* Part of source code. */
#pragma language=default
```

Example 3

Around a particular part of the source code—normally in a system header file—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 190 and *--strict*, page 203.

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 section name; cannot be a section 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 `__no_init`. Alternatively, the directive can take a string specifying a section for placing either a variable or a function whose declaration follows the pragma directive.

Example

```
#pragma location=0x5231
__no_init volatile char UART1_DR; /* UART1_DR is located at */
                                  /* address 0x5231 */

#pragma location="foo"
char fooVar; /* fooVar is located in section foo */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
...
FLASH int i; /* i is placed in the FLASH section */
```

See also

Controlling data and function placement in memory, page 150.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to the standard output stream.

Description

Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[,object_attribute,...]
```

Parameters

For a list of object attributes that can be used with this pragma directive, see *Object attributes*, page 240.

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 237.

optimize

Syntax

```
#pragma optimize=param[ param...]
```

Parameters

`balanced|size|speed`	Optimizes balanced between speed and size, optimizes for size, or optimizes for speed
`none|low|medium|high`	Specifies the level of optimization
`no_code_motion`	Turns off code motion
`no_cse`	Turns off common subexpression elimination
`no_inline`	Turns off function inlining
`no_tbaa`	Turns off type-based alias analysis
`no_unroll`	Turns off loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `speed`, `size`, and `balanced` only have effect on the `high` optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=speed
int small_and_used_often()
{
    ...
}

#pragma optimize=size
int big_and_seldom_used()
{
    ...
}
```

__printf_args

Syntax

```
#pragma __printf_args
```

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __printf_args
int printf(char const *,...);


/* Function call */
printf("%d",x);  /* Compiler checks that x is an integer */
```

required

Syntax

`#pragma required=`*`symbol`*

Parameters

symbol	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the section 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 91.

__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 printf(char const *,...);


/* Function call */
scanf("%d",x);  /* Compiler checks that x is an integer */
```

section

Syntax

`#pragma section="NAME" [__memoryattribute] [align]`

alias

`#pragma segment="NAME" [__memoryattribute] [align]`

Parameters

NAME	The name of the section or segment
__memoryattribute	An optional memory attribute identifying the memory the section will be placed in; if not specified, default memory is used.
align	Specifies an alignment for the section. The value must be a constant integer expression to the power of two.

Description

Use this pragma directive to define a section name that can be used by the section operators `__section_begin`, `__section_end`, and `__section_size`. All section declarations for a specific section must have the same memory type attribute and alignment.

If an optional memory attribute is used, the return type of the section operators `__section_begin` and `__section_end` is:

`void __memoryattribute *`.

Example

```
#pragma section="MYHUGE" __huge 4
```

See also

For more information about sections, see the chapter *Linking your application*.

STDC CX_LIMITED_RANGE

Syntax `#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}`

Parameters

`ON`	Normal complex mathematic formulas can be used.
`OFF`	Normal complex mathematic formulas cannot be used.
`DEFAULT`	Sets the default behavior, that is `OFF`.

Description

Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for `x` (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 a list of type attributes that can be used with this pragma directive, see *Type attributes*, page 237.

Description
Use this pragma directive to specify IAR-specific *type attribute*s, which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example
In this example, an `int` object with the memory attribute `__near` is defined:

```
#pragma type_attribute=__near
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__near int x;
```

See also
See the chapter *Extended keywords* for more details.

vector

Syntax

```
#pragma vector=vector1[, vector2, vector3, ...]
```

Parameters

vector	The vector number(s) of an interrupt function.

Description
Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example!

```
#pragma vector=0x14
__interrupt void my_handler(void);
```

weak

Syntax

```
#pragma weak symbol1={symbol2}
```

Parameters

symbol1	A function or variable with external linkage.
symbol2	A defined function or variable.

Description

This pragma directive can be used in one of two ways:

- To make the definition of a function or variable with external linkage a weak definition. The `__weak` attribute can also be used for this purpose.
- To create a weak alias for another function or variable. You can make more than one alias for the same function or variable.

Example

To make the definition of `foo` a weak definition, write:

```
#pragma weak foo
```

To make `NMI_Handler` a weak alias for `Default_Handler`, write:

```
#pragma weak NMI_Handler=Default_Handler
```

If `NMI_Handler` is not defined elsewhere in the program, all references to `NMI_Handler` will refer to `Default_Handler`.

See also

__weak, page 247.

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
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__get_interrupt_state`	Returns the interrupt state
`__halt`	Inserts a `HALT` instruction
`__no_operation`	Inserts a `NOP` instruction
`__set_interrupt_state`	Restores the interrupt state
`__trap`	Inserts a `TRAP` instruction
`__wait_for_exception`	Inserts a `WFE` instruction
`__wait_for_interrupt`	Inserts a `WFI` instruction

Table 37: Intrinsic functions summary

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts by inserting the `SIM` instruction.

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts by inserting the `RIM` instruction.

__get_interrupt_state

Syntax

```
__istate_t __get_interrupt_state(void);
```

Description

Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

Example

```
#include "intrinsics.h"

void CriticalFn()
{
  __istate_t s = __get_interrupt_state();
  __disable_interrupt();

  /* Do something here. */

  __set_interrupt_state(s);
}
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

__halt

Syntax
```
void __halt(void);
```

Description

Inserts a HALT instruction.

__no_operation

Syntax
```
void __no_operation(void);
```

Description

Inserts a NOP instruction.

__set_interrupt_state

Syntax
```
void __set_interrupt_state(__istate_t);
```

Descriptions

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 266.

__trap

Syntax
```
void __trap(void);
```

Description

Inserts a TRAP instruction.

__wait_for_exception

Syntax
```
void __wait_for_exception(void);
```

Description

Inserts a WFE instruction.

__wait_for_interrupt

Syntax
```
void __wait_for_interrupt(void);
```

Description

Inserts a WFI 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 STM8 adheres to Standard C. The compiler also makes these preprocessor-related features available to you:

- Predefined preprocessor symbols

  These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 270.
- 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 184.
- Preprocessor extensions

  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 272.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 200.

To specify a path for an include file, use forward slashes:

```
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile","rt");
```

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.

Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

Predefined symbol	Identifies
`__BASE_FILE__`	A string that identifies the name of the base source file (that is, not the header file), being compiled. See also *__FILE__, page 271*, and *--no_path_in_file_macros*, page 195.
`__BUILD_NUMBER__`	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__`	An integer that identifies the code model in use. The symbol reflects the `--code_model` option and is defined to `__SMALL_CODE_MODEL__`, `__MEDIUM_CODE__MODEL__`, or `__LARGE_CODE_MODEL__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.
`__CORE__`	An integer that identifies the chip core in use. The symbol reflects the `--core` option and is defined to `__STM8__`. This symbolic name can be used when testing the `__CORE__` symbol.
`__cplusplus`	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.*
`__DATA_MODEL__`	An integer that identifies the data model in use. The symbol reflects the `--data_model` option and is defined to `__SMALL_DATA_MODEL__`, `__MEDIUM_DATA_MODEL__`, or `__LARGE_DATA_MODEL__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.
`__DATE__`	A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2010`". *

Table 38: Predefined symbols

Predefined symbol	Identifies
`__embedded_cplusplus`	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.*
`__FILE__`	A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also *__BASE_FILE__*, page 270, and *–no_path_in_file_macros*, page 195.*
`__func__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see *-e*, page 190. See also *__PRETTY_FUNCTION__*, page 271.
`__FUNCTION__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see *-e*, page 190. See also *__PRETTY_FUNCTION__*, page 271.
`__IAR_SYSTEMS_ICC__`	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.
`__ICCSTM8__`	An integer that is set to `1` when the code is compiled with the IAR C/C++ Compiler for STM8.
`__LINE__`	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.*
`__LITTLE_ENDIAN__`	An integer that is defined to `0` because the STM8 byte order is big-endian.
`__PRETTY_FUNCTION__`	A string that identifies 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 *-e*, page 190. See also *__func__*, page 271.

Table 38: Predefined symbols (Continued)

Predefined symbol	Identifies
`__STDC__`	An integer that is set to `1`, which means the compiler adheres to Standard C. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to Standard C.*
`__STDC_VERSION__`	An integer that identifies the version of the C standard in use. The symbol expands to `199901L`,unless the `--c89` compiler option is used in which case the symbol expands to `199409L`. This symbol does not apply in EC++ mode.*
`__SUBVERSION__`	An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.
`__TIME__`	A string that identifies the time of compilation in the form `"hh:mm:ss"`.*
`__VER__`	An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: `(100 * the major version number + the minor version number)`. For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

Table 38: Predefined symbols (Continued)

* **This symbol is required by Standard C.**

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 91.

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 IAR DLIB Library is a complete library, compliant with Standard C and C++. It supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic project configuration*, page 19. The linker will include only those routines that are required—directly or indirectly—by your application.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- The miscellaneous functions `setlocale`, `rand`, `atexit`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `printf`, `sprintf`, `scanf`, `scanf`, `getchar`, and `putchar`.

Some functions also share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.

- Intrinsic functions, allowing low-level use of STM8 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 280.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions.

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
`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)

Table 39: Traditional Standard C header files—DLIB

Header file	Usage
wchar.h	Support for wide characters
wctype.h	Classifying wide characters

Table 39: Traditional Standard C header files—DLIB (Continued)

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

This table lists the Embedded C++ header files:

Header file	Usage
complex	Defining a class that supports complex arithmetic
exception	Defining several functions that control exception handling
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
stdexcept	Defining several classes useful for reporting exceptions
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 40: Embedded C++ header files

Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

Header file	Description
algorithm	Defines several common operations on sequences

Table 41: Standard template library header files

Header file	Description
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
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 41: 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
ccomplex	Computing common complex mathematical functions
cctype	Classifying characters
cerrno	Testing error codes reported by library functions
cfenv.h	Floating-point exception flags
cfloat	Testing floating-point type properties
cinttypes	Defining formatters for all types defined in `stdint.h`
ciso646	Using Amendment 1—iso646.h standard header
climits	Testing integer type properties

Table 42: New Standard C header files—DLIB

Header file	Usage
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
ctgmath.h	Type-generic mathematical functions
ctime	Converting between various time and date formats
cwchar	Support for wide characters
cwctype	Classifying wide characters

Table 42: 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

fenv.h

In fenv.h, trap handling support for floating-point numbers is defined with the functions fegettrapenable and fegettrapdisable. No floating-point status flags are supported.

stdio.h

These functions provide additional I/O functionality:

`fdopen`	Opens a file based on a low-level file descriptor.
`fileno`	Gets the low-level file descriptor from the file descriptor (`FILE*`).
`__gets`	Corresponds to `fgets` on `stdin`.
`getw`	Gets a `wchar_t` character from `stdin`.
`putw`	Puts a `wchar_t` character to `stdout`.
`__ungetchar`	Corresponds to `ungetc` on `stdout`.
`__write_array`	Corresponds to `fwrite` on `stdout`.

string.h

These are the additional functions defined in `string.h`:

`strdup`	Duplicates a string on the heap.
`strcasecmp`	Compare strings case-insensitive.
`strncasecmp`	Compares strings case-insensitive and bounded.
`strnlen`	Bounded string length.

The linker configuration file

This chapter describes the purpose of the linker configuration file and describes its contents.

To read this chapter you must be familiar with the concept of *sections*, see *Modules and sections*, page 43.

Overview

To link and locate an application in memory according to your requirements, ILINK needs information about how to handle sections and how to place them into the available memory regions. In other words, ILINK needs a *configuration*, passed to it by means of the *linker configuration file*.

This file consists of a sequence of directives and typically, provides facilities for:

- Defining available addressable memories

  giving the linker information about the maximum size of possible addresses and defining the available physical memory, as well as dealing with memories that can be addressed in different ways.
- Defining the regions of the available memories that are populated with ROM (flash memory or EEPROM) or RAM

  giving the start and end address for each region.
- Section groups

  dealing with how to group sections into blocks and overlays depending on the section requirements.
- Defining how to handle initialization of the application

  giving information about which sections that are to be initialized, and how that initialization should be made.
- Memory allocation

  defining where—in what memory region—each set of sections should be placed.
- Using symbols, expressions, and numbers

  expressing addresses and sizes, etc, in the other configuration directives. The symbols can also be used in the application itself.

- Structural configuration

  meaning that you can include or exclude directives depending on a condition, and to split the configuration file into several different files.

Comments can be written either as C comments (`/*...*/`) or as C++ comments (`//...`).

Defining memories and regions

ILINK needs information about the available memory spaces, or more specifically it needs information about:

- The maximum size of possible addressable memories

  The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. See *Define memory directive*, page 284.

- Available physical memory

  The `define region` directive defines a region in the available memories in which specific sections of application code and sections of application data can be placed. See *Define region directive*, page 285.

  A region consists of one or several memory ranges. A range is a continuous sequence of bytes in a memory and several ranges can be expressed by using region expressions. See *Region expression*, page 287.

Define memory directive

Syntax

```
define memory [ name ] with size = size_expr [ ,unit-size ];
```

where *`unit-size`* is one of:

```
unitbitsize = bitsize_expr
unitbytesize = bytesize_expr
```

and where *`expr`* is an expression, see *Expressions*, page 301.

Parameters

`size_expr`	Specifies how many *units* the memory space contains; always counted from address zero.
`bitsize_expr`	Specifies how many bits each unit contains.
`bytesize_expr`	Specifies how many bytes each unit contains. Each byte contains 8 bits.

Description

The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. This sets the limits for the possible addresses to be used in the linker configuration file. For many microcontrollers, one memory space is sufficient. However, some microcontrollers require two or more. For example, a Harvard architecture usually requires two different memory spaces, one for code and one for data. If only one memory is defined, the memory name is optional. If no *unit-size* is given, the unit contains 8 bits.

Example

```
/* Declare the memory space Mem of four Gigabytes */
define memory Mem with size = 4G;
```

Define region directive

Syntax

```
define region name = region-expr;
```

where *region-expr* is a region expression, see also *Regions*, page 285.

Parameters

name	The name of the region.

Description

The `define region` directive defines a region in which specific sections of code and sections of data can be placed. A region consists of one or several memory ranges, where each memory range consists of a continuous sequence of bytes in a specific memory. Several ranges can be combined by using region expressions. Note that those ranges do not need to be consecutive or even in the same memory.

Example

```
/* Define the 0x10000-byte code region ROM located at address
   0x10000 in memory Mem */
define region ROM = Mem:[from 0x10000 size 0x10000];
```

Regions

A *region* is s a set of non-overlapping memory ranges. A *region expression* is built up out of *region literals* and set operations (union, intersection, and difference) on regions.

Region literal

Syntax

```
[ memory-name: ][from expr { to expr | size expr }
  [ repeat expr [ displacement expr ]]]
```

where *expr* is an expression, see *Expressions*, page 301.

Parameters

memory-name	The name of the memory space in which the region literal will be located. If there is only one memory, the name is optional.
from	The start address of the memory range (inclusive).
to	The end address of the memory range (inclusive).
size	The size of the memory range.
repeat	Defines several ranges in the same memory for the region literal.
displacement	Displacement from the previous range start in the repeat sequence. Default displacement is the same value as the range size.

Description

A region literal consists of one memory range. When you define a range, the memory it resides in, a start address, and a size must be specified. The range size can be stated explicitly by specifying a size, or implicitly by specifying the final address of the range. The final address is included in the range and a zero-sized range will only contain an address. A range can span over the address zero and the range can even be expressed by unsigned values, because it is known where the memory wraps.

The `repeat` parameter will create a region literal that contains several ranges, one for each repeat. This is useful for *banked* or *far* regions.

Example

```
/* The 5-byte size range spans over the address zero */
Mem:[from -2 to 2]

/* The 512-byte size range spans over zero, in a 64-Kbyte memory
*/
Mem:[from 0xFF00 to 0xFF]

/* Defining several ranges in the same memory, a repeating
   literal */
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]

/* Resulting in a region containing:
   Mem:[from 0 size 0x100]
   Mem:[from 0x1000 size 0x100]
   Mem:[from 0x2000 size 0x100]
*/
```

See also

Define region directive, page 285, and *Region expression*, page 287.

Region expression

Syntax

```
region-operand
 | region-expr | region-operand
 | region-expr - region-operand
 | region-expr & region-operand
```

where `region-operand` is one of:

```
( region-expr )
region-name
region-literal
empty-region
```

where `region-name` is a region, see *Define region directive*, page 285

where `region-literal` is a region literal, see *Region literal*, page 285

and where `empty-region` is an empty region, see *Empty region*, page 288.

Description

Normally, a region consists of one memory range, which means a *region literal* is sufficient to express it. When a region contains several ranges, possibly in different memories, it is instead necessary to use a *region expression* to express it. Region expressions are actually set expressions on sets of memory ranges.

To create region expressions, three operators are available: union (`|`), intersection (`&`), and difference (`-`). These operators work as in *set theory*. For example, if you have the sets A and B, then the result of the operators would be:

- `A | B`: all elements in either set A or set B
- `A & B`: all elements in both set A and B
- `A - B`: all elements in set A but not in B.

Example

```
/* Resulting in a range starting at 1000 and ending at 2FFF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1500 and ending at 1FFF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1000 and ending at 14FF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]
```

```
/* Resulting in two ranges. The first starting at 1000 and ending
   at 1FFF, the second starting at 2501 and ending at 2FFF.
   Both located in memory Mem */
Mem:[from 0x1000 to 0x2FFF] - Mem:[from 0x2000 to 0x24FF]
```

Empty region

Syntax

```
[ ]
```

Description

The empty region does not contain any memory ranges. If the empty region is used in a placement directive that actually is used for placing one or more sections, ILINK will issue an error.

Example

```
define region Code = Mem:[from 0 size 0x10000];
if (Banked) {
  define region Bank = Mem:[from 0x8000 size 0x1000];
} else {
  define region Bank = [];
}
define region NonBanked = Code - Bank;

/* Depending on the Banked symbol, the NonBanked region is either
   one range with 0x10000 bytes, or two ranges with 0x8000 and
   0x7000 bytes, respectively. */
```

See also

Region expression, page 287.

Section handling

Section handling describes how ILINK should handle the sections of the execution image, which means:

- Placing sections in regions

  The `place at` and `place into` directives place sets of sections with similar attributes into previously defined regions. See *Place at directive*, page 295 and *Place in directive*, page 296.

- Making sets of sections with special requirements

  The `block` directive makes it possible to create empty sections with specific sizes and alignments, sequentially sorted sections of different types, etc.

  The `overlay` directive makes it possible to create an area of memory that can contain several overlay images. See *Define block directive*, page 289, and *Define overlay directive*, page 290.

- Initializing the application

  The directives `initialize` and `do not initialize` control how the application should be started. With these directives, the application can initialize global symbols at startup, and copy pieces of code. The initializers can be stored in several ways, for example they can be compressed. See *Initialize directive*, page 291 and *Do not initialize directive*, page 294.

- Keeping removed sections

  The `keep` directive retains sections even though they are not referred to by the rest of the application, which means it is equivalent to the *root* concept in the assembler and compiler. See *Keep directive*, page 294.

Define block directive

Syntax

```
define block name
  [ with param, param... ]
{
  extended-selectors
}
[except
 {
   section_selectors
  }];
```

where *param* can be one of:

```
size = expr
maximum size = expr
alignment = expr
fixed order
```

and where the rest of the directive selects sections to include in the block, see *Section selection*, page 296.

Parameters

name	The name of the defined block.
`size`	Customizes the size of the block. By default, the size of a block is the sum of its parts dependent of its contents.
`maximum size`	Specifies an upper limit for the size of the block. An error is generated if the sections in the block do not fit.
`alignment`	Specifies a minimum alignment for the block. If any section in the block has a higher alignment than the minimum alignment, the block will have that alignment.

`fixed order`	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.

Description

The `block` directive defines a named set of sections. By defining a block you can create empty blocks of bytes that can be used, for example as stacks or heaps. Another use for the directive is to group certain types of sections, consecutive or non-consecutive. A third example of use for the directive is to group sections into one memory area to access the start and end of that area from the application.

Example

```
/* Create a 0x1000-byte block for the heap */
define block HEAP with size = 0x1000, alignment = 8 { };
```

See also

Interaction between the tools and your application, page 140. See *Define overlay directive*, page 290 for an *accessing* example.

Define overlay directive

Syntax

```
define overlay name [ with param, param... ]
{
  extended-selectors;
}
[except
 {
   section_selectors
  }];
```

For information about extended selectors and except clauses, see *Section selection*, page 296.

Parameters

`name`	The name of the overlay.
`size`	Customizes the size of the overlay. By default, the size of a overlay is the sum of its parts dependent of its contents.
`maximum size`	Specifies an upper limit for the size of the overlay. An error is generated if the sections in the overlay do not fit.
`alignment`	Specifies a minimum alignment for the overlay. If any section in the overlay has a higher alignment than the minimum alignment, the overlay will have that alignment.
`fixed order`	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.

Description

The `overlay` directive defines a named set of sections. In contrast to the `block` directive, the overlay directive can define the same name several times. Each definition will then be grouped in memory at the same place as all other definitions of the same name. This creates an *overlaid* memory area, which can be useful for an application that has several independent sub-applications.

Place each sub-application image in ROM and reserve a RAM overlay area that can hold all sub-applications. To execute a sub-application, first copy it from ROM to the RAM overlay. Note that ILINK does not help you with managing interdependent overlay definitions, apart from generating a diagnostic message for any reference from one overlay to another overlay.

The size of an overlay will be the same size as the largest definition of that overlay name and the alignment requirements will be the same as for the definition with the highest alignment requirements.

Note: Sections that were overlaid must be split into a RAM and a ROM part and you must take care of all the copying needed.

See also

Manual initialization, page 59.

Initialize directive

Syntax

```
initialize { by copy | manually }
   [ with param, param... ]
{
  section-selectors
}
[except
 {
   section_selectors
  }];
```

where *param* is one of:

```
packing = { none | zeros | packbits | bwt | lzw | auto |
            smallest }
```

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 296.

Parameters

`by copy`	Splits the section into sections for initializers and initialized data, and handles the initialization at application startup automatically.
`manually`	Splits the section into sections for initializers and initialized data. The initialization at application startup is not handled automatically.

packing	Specifies how to handle the initializers. Choose between:	
	none	Disables compression of the selected section contents. This is the default method for initialize manually.
	zeros	Compresses sequential bytes with the value zero.
	packbits	Compresses with the PackBits algorithm. This method generates good results for data with many consecutive bytes of the same value.
	bwt	Compresses with the Burrows-Wheeler algorithm. This method improves the packbits method by transforming blocks of data before they are compressed.
	lzw	Compresses with the Lempel-Ziv-Welch algorithm. This method uses a dictionary to store byte patterns in the data.
	auto	Similar to smallest, but ILINK chooses between none and packbits. This is the default method for initialize by copy.
	smallest	ILINK estimates the resulting size using each packing method (except for auto), and then chooses the packing method that produces the smallest estimated size. Note that the size of the decompressor is also included.

Description

The initialize directive splits the initialization section into one section holding the initializers and another section holding the initialized data. You can choose whether the initialization at startup should be handled automatically (initialize by copy) or whether you should handle it yourself (initialize manually).

When you use the packing method auto (default for initialize by copy) or smallest, ILINK will automatically choose an appropriate packing algorithm for the initializers and automatically revert it at the initialization process when the application starts. To override this, specify a different packing method. Use the copy routine parameter to override the method for copying the initializers. The --log initialization option shows how ILINK decided which packing algorithm to use.

When initializers are compressed, a decompressor is automatically added to the image. The decompressors for `bwt` and `lzw` use significantly more execution time and RAM than `zeros` and `packbits`. Approximately 9 Kbytes of stack space is needed for `bwt` and 3.5 Kbytes for `lzw`.

When initializers are compressed, the exact size of the compressed initializers is unknown until the exact content of the uncompressed data is known. If this data contains other addresses, and some of these addresses are dependent on the size of the compressed initializers, the linker fails with error Lp017. To avoid this, place compressed initializers last, or in a memory region together with sections whose addresses do not need to be known.

Optionally, ILINK will also produce a table that an initialization function in the system startup code uses for copying the section contents from the initializer sections to the corresponding original sections. Normally, the section content is initialized variables.

Zero-initialized sections are not affected by the `initialize` directive.

Sections that must execute before the initialization finishes are not affected by the `initialize by copy` directive. This includes the `__low_level_init` function and anything it references.

Anything reachable from the program entry label is considered *needed for initialization* unless reached via a section fragment with a label starting with `__iar_init$$done`. The `--log sections` option can be used for creating a log of this process (in addition to the more general process of marking section fragments to be included in the application).

The `initialize` directive can be used for copying other things as well, for example copying executable code from slow ROM to fast RAM. For another example, see *Define overlay directive*, page 290.

Example

```
/* Copy all read-write sections automatically from ROM to RAM at
   program start */
initialize by copy { rw };
place in RAM { rw };
place in ROM { ro };
```

See also

Initialization at system startup, page 49, and *Do not initialize directive*, page 294.

Do not initialize directive

Syntax

```
do not initialize
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 296.

Description

The `do not initialize` directive specifies the sections that should not be initialized by the system startup code. The directive can only be used on `zeroinit` sections.

The compiler keyword `__no_init` places variables into sections that must be handled by a `do not initialize` directive.

Example

```
/* Do not initialize read-write sections whose name ends with
   _noinit at program start */
do not initialize { rw section .*_noinit };
place in RAM { rw section .*_noinit };
```

See also

Initialization at system startup, page 49, and *Initialize directive*, page 291.

Keep directive

Syntax

```
keep
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 296.

Description

The `keep` directive specifies that all selected sections should be kept in the executable image, even if there are no references to the sections.

Example

```
keep { section .keep* } except {section .keep};
```

Place at directive

Syntax

```
[ "name": ]
place at { address [ memory: ] expr | start of region_expr |
           end of region_expr }
{
  extended-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 296.

Parameters

memory: *expr*	A specific address in a specific memory. The address must be available in the supplied memory defined by the `define memory` directive. The memory specifier is optional if there is only one memory.
start of *region_expr*	A region expression. The start of the region is used.
end of *region_expr*	A region expression. The end of the region is used.

Description

The `place at` directive places sections and blocks either at a specific address or, at the beginning or the end of a region. The same address cannot be used for two different `place at` directives. It is also not possible to use an empty region in a `place at` directive. If placed in a region, the sections and blocks will be placed before any other sections or blocks placed in the same region with a `place in` directive.

The sections and blocks will be placed in the region in an arbitrary order. To specify a specific order, use the `block` directive.

The *name*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only section .startup at the beginning of the
   code_region */
"START": place at start of ROM { readonly section .startup };
```

See also

Place in directive, page 296.

Place in directive

Syntax

```
[ "name": ]
place in region-expr
{
  extended-selectors
}
 [{
   section-selectors
 }];
```

where `region-expr` is a region expression, see also *Regions*, page 285.

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 296.

Description

The `place in` directive places sections and blocks in a specific region. The sections and blocks will be placed in the region in an arbitrary order.

To specify a specific order, use the `block` directive. The region can have several ranges.

The *`name`*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only sections in the code_region */
"ROM": place in ROM { readonly };
```

See also

Place at directive, page 295.

Section selection

The purpose of *section selection* is to specify—by means of *section selectors* and *except clauses*—the sections that an ILINK directive should be applied to. All sections that match one or more of the section selectors will be selected, and none of the sections selectors in the except clause, if any. Each section selector can match sections on section attributes, section name, and object or library name.

Some directives provide functionality that requires more detailed selection capabilities, for example directives that can be applied on both sections and blocks. In this case, the *extended-selectors* are used.

Section-selectors

Syntax

```
{ [ section-selector ][, section-selector...] }
```

where *section-selector* is:

```
[ section-attribute ][ section-type ][ section sectionname ]
    [object {module | filename} ]
```

where *section-attribute* is:

```
[ ro [ code | data ] | rw [ code | data ] | zi ]
```

and where `ro`, `rw`, and `zi` also can be `readonly`, `readwrite`, and `zeroinit`, respectively.

And *section-type* is:

```
[ preinit_array | init_array ]
```

Parameters

`ro` or `readonly`	Read-only sections.
`rw` or `readwrite`	Read/write sections.
`zi` or `zeroinit`	Zero-initialized sections. These sections should be initialized with zeros during system startup.
`code`	Sections that contain code.
`data`	Sections that contain data.
`preinit_array`	Sections of the ELF section type `SHT_PREINIT_ARRAY`.
`init_array`	Sections of the ELF section type `SHT_INIT_ARRAY`.
`sectionname`	The section name. Two wildcards are allowed: `?` matches any single character `*` matches zero or more characters.
`module`	A name in the form *`objectname(libraryname)`*. Sections from object modules where both the object name and the library name match their respective patterns are selected. An empty library name pattern selects only sections from object files.
`filename`	The name of an object file, a library, or an object in a library. Two wildcards are allowed: `?` matches any single character `*` matches zero or more characters.

Description

A section selector selects all sections that match the section attribute, section type, section name, and the name of the *object*, where *object* is an object file, a library, or an

object in a library. Only up to three of the four conditions can be omitted. If the section attribute is omitted, any section will be selected, without restrictions on the section attribute. If the section type is omitted, sections of any type will be selected.

If the section name part or the object name part is omitted, sections will be selected without restrictions on the section name or object name, respectively.

It is also possible to use only `{ }` without any section selectors, which can be useful when defining blocks.

Note that a section selector with narrower scope has higher priority than a more generic section selector.

If more than one section selector matches for the same purpose, one of them must be more specific. A section selector is more specific than another one if:

- It specifies a section type and the other one does not
- It specifies a section name or object name with no wildcards and the other one does not
- There could be sections that match the other selector that also match this one, and the reverse is not true.

Selector 1	Selector 2	More specific
`section "foo*"`	`section "f*"`	Selector 1
`section "*x"`	`section "f*"`	Neither
`ro code section "f*"`	`ro section "f*"`	Selector 1
`init_array`	`ro section "xx"`	Selector 1
`section ".intvec"`	`ro section ".int*"`	Selector 1
`section ".intvec"`	`object "foo.o"`	Neither

Table 43: Examples of section selector specifications

Example

```
{ rw }                      /* Selects all read-write sections */

{ section .mydata* }        /* Selects only .mydata* sections */
/* Selects .mydata* sections available in the object special.o */
{ section .mydata* object special.o }
```

Assuming a section in an object named `foo.o` in a library named `lib.a`, any of these selectors will select that section:

```
object foo.o(lib.a)
object f*(lib*)
object foo.o
object lib.a
```

See also

Initialize directive, page 291, *Do not initialize directive*, page 294, and *Keep directive*, page 294.

Extended-selectors

Syntax

```
{ [ extended-selector ][, extended-selector...] }
```

where *extended-selector* is:

```
[ first | last ]{ section-selector |
                  block name [ inline-block-def ]|
                  overlay name }
```

where *inline-block-def* is:

```
[ block-params ] extended-selectors
```

Parameters

`first`	Places the selected name first in the region, block, or overlay.
`last`	Places the selected name last in the region, block, or overlay.
`block`	The name of the block.
`overlay`	The name of the overlay.

Description

In addition to what the `section-selector` does, `extended-selector` provides functionality for placing blocks or overlays first or last in a set of sections, a block, or an overlay. It is also possible to create an *inline* definition of a block. This means that you can get more precise control over section placement.

Example

```
define block First { section .first }; /* Define a block holding
                                          the section .first */
define block Table { first block First }; /* Define a block where
                                             First is placed
                                             first */
```

or, equivalently using an inline definition of the block `First`:

```
define block Table { first block First { section .first }};
```

See also

Define block directive, page 289, *Define overlay directive*, page 290, and *Place at directive*, page 295.

Using symbols, expressions, and numbers

In the linker configuration file, you can also:

- Define and export symbols

  The `define symbol` directive defines a symbol with a specified value that can be used in expressions in the configuration file. The symbol can also be exported to be used by the application or the debugger. See *Define symbol directive*, page 300, and *Export directive*, page 301.

- Use expressions and numbers

  In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, et cetera. See *Expressions*, page 301.

Define symbol directive

Syntax

```
define [ exported ] symbol name = expr;
```

Parameters

`exported`	Exports the symbol to be usable by the executable image.
`name`	The name of the symbol.
`expr`	The symbol value.

Description

The `define symbol` directive defines a symbol with a specified value. The symbol can then be used in expressions in the configuration file. The symbols defined in this way work exactly like the symbols defined with the option `--config_def` outside of the configuration file.

The `define exported symbol` variant of this directive is a shortcut for using the directive `define symbol` in combination with the `export symbol` directive. On the command line this would require both a `--config_def` option and a `--define_symbol` option to achieve the same effect.

Note:

- A symbol cannot be redefined
- Symbols that are either prefixed by _*X*, where *X* is a capital letter, or that contain __ (double underscore) are reserved for toolset vendors.

Example

```
/* Define the symbol my_symbol with the value 4 */
define symbol my_symbol = 4;
```

See also

Export directive, page 301 and *Interaction between ILINK and the application*, page 61.

Export directive

Syntax

```
export symbol name;
```

Parameters

name	The name of the symbol.

Description

The `export` directive defines a symbol to be exported, so that it can be used both from the executable image and from a global label. The application, or the debugger, can then refer to it for setup purposes etc.

Example

```
/* Define the symbol my_symbol to be exported */
export symbol my_symbol;
```

Expressions

Syntax

An expression is built up of the following constituents:

```
expression binop expression
unop expression
expression ? expression : expression
(expression)
number
symbol
func-operator
```

where *binop* is one of these binary operators:

`+, -, *, /, %, <<, >>, <, >, ==, !=, &, ^, |, &&, ||`

where `unop` is one of this unary operators:

`+, -, !, ~`

where *number* is a number, see *Numbers*, page 302

where *symbol* is a defined symbol, see *Define symbol directive*, page 300 and --*config_def*, page 209

and where *func-operator* is one of these function-like operators:

`minimum(`*expr*`,`*expr*`)`	Returns the smallest of the two parameters.
`maximum(`*expr*`,`*expr*`)`	Returns the largest of the two parameters.
`isempty(`*r*`)`	Returns True if the region is empty, otherwise False.

`isdefinedsymbol(`*`expr-symbol`*`)`	Returns True if the expression symbol is defined, otherwise False.
`start(`*`r`*`)`	Returns the lowest address in the region.
`end(`*`r`*`)`	Returns the highest address in the region.
`size(`*`r`*`)`	Returns the size of the complete region.

where *`expr`* is an expression, and *`r`* is a region expression, see *Region expression*, page 287.

Description

In the linker configuration file, an expression is a 65-bit value with the range -2^64 to 2^64. The expression syntax closely follows C syntax with some minor exceptions. There are no assignments, casts, pre- or post-operations, and no address operations (`*`, `&`, `[]`, `->`, and `.`). Some operations that extract a value from a region expression, etc, use a syntax resembling that of a function call. A boolean expression returns 0 (false) or 1 (true).

Numbers

Syntax

`nr` `[`*`nr-suffix`*`]`

where `nr` is either a decimal number or a hexadecimal number (`0x...` or `0X...`).

and where *`nr-suffix`* is one of:

```
K          /* Kilo = (1 << 10) 1024 */
M          /* Mega = (1 << 20) 1048576 */
G          /* Giga = (1 << 30) 1073741824 */
T          /* Tera = (1 << 40) 1099511627776 */
P          /* Peta = (1 << 50) 1125899906842624 */
```

Description

A number can be expressed either by normal C means or by suffixing it with a set of useful suffixes, which provides a compact way of specifying numbers.

Example

`1024` is the same as `0x400`, which is the same as `1K`.

Structural configuration

The structural directives provide means for creating structure within the configuration, such as:

- Conditional inclusion

  An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations in the same file. See *If directive*, page 303.
- Dividing the linker configuration file into several different files

  The `include` directive makes it possible to divide the configuration file into several logically distinct files. See *Include directive*, page 304.

If directive

Syntax

```
if (expr) {
  directives
[ } else if (expr) {
  directives ]
[ } else {
  directives ]
}
```

where *expr* is an expression, see *Expressions*, page 301.

Parameters

directives	Any ILINK directive.

Description

An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations, for example both a banked and non-banked memory configuration, in the same file.

The directives inside an `if` part, `else if` part, or an `else` part are syntax checked and processed regardless of whether the conditional expression was true or false, but only the directives in the part where the conditional expression was true, or the `else` part if none of the conditions were true, will have any effect outside the `if` directive. The `if` directives can be nested.

Example

See *Empty region*, page 288.

Include directive

Syntax `include filename;`

Parameters

filename	A string literal where both / and \ can be used as the directory delimiter.

Description

The `include` directive makes it possible to divide the configuration file into several logically distinct files. For instance, files that you need to change often and files that you seldom edit.

Section reference

The compiler places code and data into sections. Based on a configuration specified in the linker configuration file, ILINK places sections in memory.

This chapter lists all predefined sections and blocks that are available for the IAR build tools for STM8, and gives detailed reference information about each section.

For more information about sections, see the chapter *Modules and sections*, page 43.

Summary of sections

This table lists the sections and blocks that are used by the IAR build tools:

Section	Description
`CSTACK`	Holds the stack used by C or C++ programs.
`.difunct`	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before `main` is called.
`.eeprom.noinit`	Holds `__no_init __eeprom` static and global variables.
`.far.bss`	Holds zero-initialized `__far` static and global variables.
`.far.data`	Holds `__far` static and global initialized variables.
`.far.data_init`	Holds initial values for `.far.data` sections, when the linker directive `initialize by copy` is used.
`.far.noinit`	Holds `__no_init __far` static and global variables.
`.far.rodata`	Holds `__far` constant data.
`.far_func.text`	Holds `__far_func` program code.
`HEAP`	Holds the heap used for dynamically allocated data.
`.huge.bss`	Holds zero-initialized `__huge` static and global variables.
`.huge.data`	Holds `__huge` static and global initialized variables.
`.huge.data_init`	Holds initial values for `.huge.data` sections, when the linker directive `initialize by copy` is used.
`.huge.noinit`	Holds `__no_init __huge` static and global variables.
`.huge.rodata`	Holds `__huge` constant data.

Table 44: Section summary

Section	Description
.huge_func.text	Holds __huge_func program code.
.iar.dynexit	Holds the atexit table.
.intvec	Holds the interrupt vector table, including the reset vector.
.near.bss	Holds zero-initialized __near static and global variables.
.near.data	Holds __near static and global initialized variables.
.near.data_init	Holds initial values for .near.data sections, when the linker directive initialize by copy is used.
.near.noinit	Holds __no_init __near static and global variables.
.near.rodata	Holds __near constant data.
.near_func.text	Holds __near_func program code.
.tiny.bss	Holds zero-initialized __tiny static and global variables.
.tiny.data	Holds __tiny static and global initialized variables.
.tiny.data_init	Holds initial values for .tiny.data sections, when the linker directive initialize by copy is used.
.tiny.noinit	Holds __no_init __tiny static and global variables.
.tiny.rodata	Holds __tiny constant data.
.tiny.rodata_init	Holds initial values for .tiny.rodata_init sections, when the linker directive initialize by copy is used.
.vregs	Holds virtual registers used for temporary storage.

Table 44: Section summary (Continued)

In addition to the ELF sections used for your application, the tools use a number of other ELF sections for a variety of purposes:

- Sections starting with .debug generally contain debug information in the DWARF format
- Sections starting with .iar.debug contain supplemental debug information in an IAR format
- The section .comment contains the tools and command lines used for building the file
- Sections starting with .rel or .rela contain ELF relocation information
- The section .symtab contains the symbol table for a file
- The section .strtab contains the names of the symbol in the symbol table
- The section .shstrtab contains the names of the sections.

Descriptions of sections and blocks

This section gives reference information about each section, where the:

- *Description* describes what type of content the section is holding and, where required, how the section is treated by the linker
- *Memory placement* describes memory placement restrictions.

For information about how to allocate sections in memory by modifying the linker configuration file, see the *Placing code and data—the linker configuration file*, page 46.

CSTACK

Description
Block that holds the internal data stack.

Memory placement
This block must be placed in RAM in the first 64 Kbytes of memory.

See also
Setting up the stack, page 58.

.difunct

Description
Holds the dynamic initialization vector used by C++.

Memory placement
In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

.eeprom.noinit

Description
Holds static and global `__no_init __eeprom` variables.

Memory placement
This section must be placed in the EEPROM memory, which must be a part of the first 64 Kbytes of memory.

See also
Memory types, page 26.

.far.bss

Description
Holds zero-initialized `__far` static and global variables.

Memory placement
In RAM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also
Memory types, page 26.

.far.data

Description: Holds `__far` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize by copy` is used, a corresponding `.far.data_init` section is created for each `.far.data` section, holding the possibly compressed initial values.

Memory placement: In RAM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also: *Memory types*, page 26.

.far.data_init

Description: Holds possibly compressed initial values for `.far.data` sections. This section is created by the linker if the `initialize by copy` linker directive is used.

Memory placement: In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also: *Memory types*, page 26.

.far.noinit

Description: Holds static and global `__no_init __far` variables.

Memory placement: In RAM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also: *Memory types*, page 26.

.far.rodata

Description: Holds `__far` constant data. This can include constant variables, string and aggregate literals, etc.

Memory placement: In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also: *Memory types*, page 26.

.far_func.text

Description — Holds `__far_func` program code.

Memory placement — In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also — *Using function memory attributes*, page 36.

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`.

Memory placement — This section must be placed in RAM in the first 64 Kbytes of memory.

See also — *Setting up the heap*, page 58.

.huge.bss

Description — Holds zero-initialized `__huge` static and global variables.

Memory placement — In RAM anywhere in the 16 Mbytes of memory.

See also — *Memory types*, page 26.

.huge.data

Description — Holds `__huge` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize by copy` is used, a corresponding `.huge.data_init` section is created for each `.huge.data` section, holding the possibly compressed initial values.

Memory placement — In RAM anywhere in the 16 Mbytes of memory.

See also — *Memory types*, page 26.

.huge.data_init

Description: Holds possibly compressed initial values for `.huge.data` sections. This section is created by the linker if the `initialize by copy` linker directive is used.

Memory placement: In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also: *Memory types*, page 26.

.huge.noinit

Description: Holds static and global `__no_init __huge` variables.

Memory placement: In RAM anywhere in the 16 Mbytes of memory.

See also: *Memory types*, page 26.

.huge.rodata

Description: Holds `__huge` constant data. This can include constant variables, string and aggregate literals, etc.

Memory placement: In ROM anywhere in the 16 Mbytes of memory.

See also: *Memory types*, page 26.

.huge_func.text

Description: Holds `__huge_func` program code.

Memory placement: In ROM anywhere in the 16 Mbytes of memory.

See also: *Using function memory attributes*, page 36.

.iar.dynexit

Description: Holds the table of calls to be made at exit.

Memory placement: In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also — *Setting up the atexit limit*, page 58.

.intvec

Description — Holds the interrupt vector table, including the reset vector.

Memory placement — This section must be placed in ROM where the reset vector is located.

.near.bss

Description — Holds zero-initialized `__near` static and global variables.

Memory placement — In RAM in the first 64 Kbytes of memory.

See also — *Memory types*, page 26.

.near.data

Description — Holds `__near` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize by copy` is used, a corresponding `.near.data_init` section is created for each `.near.data` section, holding the possibly compressed initial values.

Memory placement — In RAM in the first 64 Kbytes of memory.

See also — *Memory types*, page 26.

.near.data_init

Description — Holds possibly compressed initial values for `.near.data` sections. This section is created by the linker if the `initialize by copy` linker directive is used.

Memory placement — In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also — *Memory types*, page 26.

.near.noinit

Description — Holds static and global `__no_init __near` variables.

Memory placement	In RAM in the first 64 Kbytes of memory.
See also	*Memory types*, page 26.

.near.rodata

Description	Holds `__near` constant data. This can include constant variables, string and aggregate literals, etc.
Memory placement	In ROM in the first 64 Kbytes of memory.
See also	*Memory types*, page 26.

.near_func.text

Description	Holds `__near_func` program code.
Memory placement	In ROM in the first 64 Kbytes of memory.
See also	*Using function memory attributes*, page 36.

.tiny.bss

Description	Holds zero-initialized `__tiny` static and global variables.
Memory placement	In RAM anywhere in the first 256 bytes of memory.
See also	*Memory types*, page 26.

.tiny.data

Description	Holds `__tiny` static and global initialized variables including initializers. In object files, this includes the initial values. When the linker directive `initialize by copy` is used, a corresponding `.tiny.data_init` section is created for each `.tiny.data` section, holding the possibly compressed initial values.
Memory placement	In RAM anywhere in the first 256 bytes of memory.
See also	*Memory types*, page 26.

.tiny.data_init

Description — Holds possibly compressed initial values for `.tiny.data` sections. This section is created by the linker if the `initialize by copy` linker directive is used.

Memory placement — In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also — *Memory types*, page 26.

.tiny.noinit

Description — Holds static and global `__no_init __tiny` variables.

Memory placement — In RAM anywhere in the first 256 bytes of memory.

See also — *Memory types*, page 26.

.tiny.rodata

Description — Holds `__tiny` constant data. As there is no ROM in tiny memory, the linker directive `initialize by copy` is used for `.tiny.rodata` sections, creating a corresponding `.tiny.rodata_init` section for each `.tiny.rodata` section, holding the actual values. This can include constant variables, string and aggregate literals, etc.

Memory placement — In ROM anywhere in the first 256 bytes of memory.

See also — *Memory types*, page 26.

.tiny.rodata_init

Description — Holds initializers for `.tiny.rodata` sections. Initializers will be copied to RAM at startup. This section is created by the linker.

Memory placement — In ROM anywhere in the 16 Mbytes of memory, but separated into 64-Kbyte sections.

See also — *Memory types*, page 26.

.vregs

Description	Holds virtual registers used by the compiler for temporary storage.
Memory placement	In RAM in the first 256 bytes of memory.

IAR utilities

This chapter describes the IAR command line utilities that are available:

- The IAR Archive Tool—iarchive—creates and manipulates a library (an archive) of several ELF object files
- The IAR ELF Tool—ielftool—performs various transformations on an ELF executable image (such as fill, checksum, format conversions, etc)
- The IAR ELF Dumper for STM8—ielfdumpstm8—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—iobjmanip—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—isymexport—exports absolute symbols from a ROM image file, so that they can be used when you link an add-on application.

The IAR Archive Tool—iarchive

The IAR Archive Tool, `iarchive`, can create a library (an archive) file from several ELF object files. You can also use `iarchaive` to manipulate ELF libraries.

A library file contains several relocatable ELF object modules, each of which can be independently used by a linker. In contrast with object modules specified directly to the linker, each module in a library is only included if it is needed.

For information about how to build a library in the IDE, see the *IAR Embedded Workbench® IDE User Guide.*

INVOCATION SYNTAX

The invocation syntax for the archive builder is:

```
iarchive parameters
```

Parameters

The parameters are:

Parameter	Description
command	Command line options that define an operation to be performed. Such an option must be specified before the name of the library file.
libraryfile	The library file to be operated on.
objectfile1 ... objectfileN	The object file(s) that the specified command operates on.
options	Command line options that define actions to be performed. These options can be placed anywhere on the command line.

Table 45: iarchive parameters

Examples

This example creates a library file called `mylibrary.a` from the source object files `module1.o`, `module.2.o`, and `module3.o`:

```
iarchive mylibrary.a module1.o module2.o module3.o.
```

This example lists the contents of `mylibrary.a`:

```
iarchive --toc mylibrary.a
```

This example replaces `module3.o` in the library with the content in the `module3.o` file and appends `module4.o` to `mylibrary.a`:

```
iarchive --replace mylibrary.a module3.o module4.o
```

SUMMARY OF IARCHIVE COMMANDS

This table summarizes the `iarchive` commands:

Command line option	Description
`--create`	Creates a library that contains the listed object files.
`--delete, -d`	Deletes the listed object files from the library.
`--extract, -x`	Extracts the listed object files from the library.
`--replace, -r`	Replaces or appends the listed object files to the library.
`--symbols`	Lists all symbols defined by files in the library.
`--toc, -t`	Lists all files in the library.

Table 46: iarchive commands summary

For detailed descriptions, see *Descriptions of options*, page 329.

SUMMARY OF IARCHIVE OPTIONS

This table summarizes the `iarchive` options:

Command line option	Description
`-f`	Extends the command line.
`--output, -o`	Specifies the library file.
`--silent`	Sets silent operation.
`--verbose, -V`	Reports all performed operations.

Table 47: iarchive options summary

For detailed descriptions, see *Descriptions of options*, page 329.

DIAGNOSTIC MESSAGES

This section lists the messages produced by `iarchive`:

La001: could not open file *filename*

`iarchive` failed to open an object file.

La002: illegal path *pathname*

The path *`pathname`* is not a valid path.

La006: too many parameters to *cmd* command

A list of object modules was specified as parameters to a command that only accepts a single library file.

La007: too few parameters to *cmd* command

A command that takes a list of object modules was issued without the expected modules.

La008: *lib* is not a library file

The library file did not pass a basic syntax check. Most likely the file is not the intended library file.

La009: *lib* has no symbol table

The library file does not contain the expected symbol information. The reason might be that the file is not the intended library file, or that it does not contain any ELF object modules.

La010: no library parameter given

The tool could not identify which library file to operate on. The reason might be that a library file has not been specified.

La011: file *file* already exists

The file could not be created because a file with the same name already exists.

La013: file confusions, *lib* given as both library and object

The library file was also mentioned in the list of object modules.

La014: module *module* not present in archive *lib*

The specified object module could not be found in the archive.

La015: internal error

The invocation triggered an unexpected error in `iarchive`.

Ms003: could not open file *filename* for writing

`iarchive` failed to open the archive file for writing. Make sure that it is not write protected.

Ms004: problem writing to file *filename*

An error occurred while writing to file *`filename`*. A possible reason for this is that the volume is full.

Ms005: problem closing file *filename*

An error occurred while closing the file *`filename`*.

The IAR ELF Tool—ielftool

The IAR ELF Tool, `ielftool`, can generate a checksum on specific ranges of memories. This checksum can be compared with a checksum calculated on your application.

The source code for `ielftool` and a Microsoft VisualStudio 2005 template project are available in the `stm8\src\elfutils` directory. If you have specific requirements for how the checksum should be generated or requirements for format conversion, you can modify the source code accordingly.

INVOCATION SYNTAX

The invocation syntax for the IAR ELF Tool is:

```
ielftool [options] inputfile outputfile [options]
```

The `ielftool` tool will first process all the fill options, then it will process all the checksum options (from left to right).

Parameters

The parameters are:

Parameter	Description
inputfile	An absolute ELF executable image produced by the ILINK linker.
options	Any of the available command line options, see *Summary of ielftool options*, page 319.
outputfile	An absolute ELF executable image.

Table 48: ielftool parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 178.

Example

This example fills a memory range with `0xFF` and then calculates a checksum on the same range:

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
           --checksum __checksum:4,crc32;0-0xFF
```

SUMMARY OF IELFTOOL OPTIONS

This table summarizes the `ielftool` command line options:

Command line option	Description
`--bin`	Sets the format of the output file to binary.
`--checksum`	Generates a checksum.
`--fill`	Specifies fill requirements.
`--ihex`	Sets the format of the output file to linear Intel hex.
`--self_reloc`	Not for general use.
`--silent`	Sets silent operation.
`--simple`	Sets the format of the output file to Simple code.
`--srec`	Sets the format of the output file to Motorola S-records.

Table 49: ielftool options summary

Command line option	Description
`--srec-len`	Restricts the number of data bytes in each S-record.
`--srec-s3only`	Restricts the S-record output to contain only a subset of records.
`--strip`	Removes debug information.
`--verbose, -V`	Prints all performed operations.

Table 49: ielftool options summary (Continued)

For detailed descriptions, see *Descriptions of options*, page 329.

The IAR ELF Dumper for STM8—ielfdumpstm8

The IAR ELF Dumper for STM8, `ielfdumpstm8`, can be used for creating a text representation of the contents of a relocatable or absolute ELF file.

`ielfdumpstm8` can be used in one of three ways:

- To produce a listing of the general properties of the input file and the ELF segments and ELF sections it contains. This is the default behavior when no command line options are used.
- To also include a textual representation of the contents of each ELF section in the input file. To specify this behavior, use the command line option `--all`.
- To produce a textual representation of selected ELF sections from the input file. To specify this behavior, use the command line option `--section`.

INVOCATION SYNTAX

The invocation syntax for `ielfdumpstm8` is:

```
ielfdumpstm8 input_file [output_file]
```

Note: `ielfdumpstm8` is a command line tool which is not primarily intended to be used in the IDE.

Parameters

The parameters are:

Parameter	Description
input_file	An ELF relocatable or executable file to use as input.
output_file	A file or directory where the output is emitted. If absent and no `--output` option is specified, output is directed to the console.

Table 50: ielfdumpstm8 parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 178.

SUMMARY OF IELFDUMPSTM8 OPTIONS

This table summarizes the `ielfdumpstm8` command line options:

Command line option	Description
`--all`	Generates output for all input sections regardless of their names or numbers.
`--code`	Dumps all sections that contain executable code.
`-f`	Extends the command line.
`--output, -o`	Specifies an output file.
`--no_strtab`	Suppresses dumping of string table sections.
`--raw`	Uses the generic hexadecimal/ASCII output format for the contents of any selected section, instead of any dedicated output format for that section.
`--section, -s`	Generates output for selected input sections.

Table 51: ielfdumpstm8 options summary

For detailed descriptions, see *Descriptions of options*, page 329.

The IAR ELF Object Tool—iobjmanip

Use the IAR ELF Object Tool, `iobjmanip`, to perform low-level manipulation of ELF object files.

INVOCATION SYNTAX

The invocation syntax for the IAR ELF Object Tool is:

```
iobjmanip options inputfile outputfile
```

Parameters

The parameters are:

Parameter	Description
`options`	Command line options that define actions to be performed. These options can be placed anywhere on the command line. At least one of the options must be specified.
`inputfile`	A relocatable ELF object file.

Table 52: iobjmanip parameters

Parameter	Description
outputfile	A relocatable ELF object file with all the requested operations applied.

Table 52: iobjmanip parameters (Continued)

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 178.

Examples

This example renames the section `.example` in `input.o` to `.example2` and stores the result in `output.o`:

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

SUMMARY OF IOBJMANIP OPTIONS

This table summarizes the `iobjmanip` options:

Command line option	Description
`-f`	Extends the command line.
`--remove_section`	Removes a section.
`--rename_section`	Renames a section.
`--rename_symbol`	Renames a symbol.
`--strip`	Removes debug information.

Table 53: iobjmanip options summary

For detailed descriptions, see *Descriptions of options*, page 329.

DIAGNOSTIC MESSAGES

This section lists the messages produced by `iobjmanip`:

Lm001: No operation given

None of the command line parameters specified an operation to perform.

Lm002: Expected *nr* parameters but got *nr*

Too few or too many parameters. Check invocation syntax for `iobjmanip` and for the used command line options.

Lm003: Invalid section/symbol renaming pattern *pattern*

The pattern does not define a valid renaming operation.

Lm004: Could not open file *filename*

`iobjmanip` failed to open the input file.

Lm005: ELF format error *msg*

The input file is not a valid ELF object file.

Lm006: Unsupported section type *nr*

The object file contains a section that `iobjmanip` cannot handle. This section will be ignored when generating the output file.

Lm007: Unknown section type *nr*

`iobjmanip` encountered an unrecognized section. `iobjmanip` will try to copy the content as is.

Lm008: Symbol *symbol* has unsupported format

`iobjmanip` encountered a symbol that cannot be handled. `iobjmanip` will ignore this symbol when generating the output file.

Lm009: Group type *nr* not supported

`iobjmanip` only supports groups of type `GRP_COMDAT`. If any other group type is encountered, the result is undefined.

Lm010: Unsupported ELF feature in *file*: *msg*

The input file uses a feature that `iobjmanip` does not support.

Lm011: Unsupported ELF file type

The input file is not a relocatable object file.

Lm012: Ambiguous rename for section/symbol *name* (*alt1* and *alt2*)

An ambiguity was detected while renaming a section or symbol. One of the alternatives will be used.

Lm013: Section *name* removed due to transitive dependency on *name*

A section was removed as it depends on an explicitly removed section.

Lm014: File has no section with index *nr*

A section index, used as a parameter to `--remove_section` or `--rename_section`, did not refer to a section in the input file.

Ms003: could not open file *filename* for writing

`iobjmanip` failed to open the output file for writing. Make sure that it is not write protected.

Ms004: problem writing to file *filename*

An error occurred while writing to file *`filename`*. A possible reason for this is that the volume is full.

Ms005: problem closing file *filename*

An error occurred while closing the file *`filename`*.

The IAR Absolute Symbol Exporter—isymexport

The IAR Absolute Symbol Exporter, `isymexport`, can export absolute symbols from a ROM image file, so that they can be used when you link an add-on application.

INVOCATION SYNTAX

The invocation syntax for the IAR Absolute Symbol Exporter is:

```
isymexport [options] inputfile outputfile [options]
```

Parameters

The parameters are:

Parameter	Description
inputfile	A ROM image in the form of an executable ELF file (output from linking).
options	Any of the available command line options, see *Summary of isymexport options*, page 325.
outputfile	A relocatable ELF file that can be used as input to linking, and which contains all or a selection of the absolute symbols in the input file. The output file contains only the symbols, not the actual code or data sections. A steering file can be used to control which symbols that are included, and also to rename some of the symbols if that is desired.

Table 54: ielftool parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 178.

SUMMARY OF ISYMEXPORT OPTIONS

This table summarizes the `isymexport` command line options:

Command line option	Description
`--edit`	Specifies a steering file.
`-f`	Extends the command line.
`--ram_reserve_ranges`	Generates symbols to reserve the areas in RAM that the image uses.
`--reserve_ranges`	Generates symbols to reserve the areas in ROM and RAM that the image uses.

Table 55: isymexport options summary

For detailed descriptions, see *Descriptions of options*, page 329.

STEERING FILES

A steering file can be used for controlling which symbols that are included, and also to rename some of the symbols if that is desired. In the file, you can use `show` and `hide` directives to select which public symbols from the input file that are to be included in the output file. `rename` directives can be used for changing the names of symbols in the input file.

Syntax

The following syntax rules apply:

- Each directive is specified on a separate line.
- C comments (`/*...*/`) and C++ comments (`//...`) can be used.
- Patterns can contain wildcard characters that match more than one possible character in a symbol name.
- The `*` character matches any sequence of zero or more characters in a symbol name.
- The `?` character matches any single character in a symbol name.

Example

```
rename xxx_* as YYY_* /*Change symbol prefix from xxx_ to YYY_ */
show YYY_*            /* Export all symbols from YYY package */
hide *_internal       /* But do not export internal symbols */
show zzz?             /* Export zzza, but not zzzaaa */
hide zzzx             /* But do not export zzzx */
```

Show directive

Syntax

`show pattern`

Parameters

pattern	A pattern to match against a symbol name.

Description

A symbol with a name that matches the pattern will be included in the output file unless this is overridden by a later `hide` directive.

Example

```
/* Include all public symbols ending in _pub. */
show *_pub
```

Hide directive

Syntax

`hide pattern`

Parameters

pattern	A pattern to match against a symbol name.

Description

A symbol with a name that matches the pattern will not be included in the output file unless this is overridden by a later `show` directive.

Example

```
/* Do not include public symbols ending in _sys. */
hide *_sys
```

Rename directive

Syntax

`rename pattern1 pattern2`

Parameters

pattern1	A pattern used for finding symbols to be renamed. The pattern can contain no more than one `*` or `?` wildcard character.
pattern2	A pattern used for the new name for a symbol. If the pattern contains a wildcard character, it must be of the same kind as in *pattern1*.

Description

Use this directive to rename symbols from the output file to the input file. No exported symbol is allowed to match more than one `rename` pattern.

`rename` directives can be placed anywhere in the steering file, but they are executed before any `show` and `hide` directives. Thus, if a symbol will be renamed, all `show` and `hide` directives in the steering file must refer to the new name.

If the name of a symbol matches a *pattern1* pattern that contains no wildcard characters, the symbol will be renamed *pattern2* in the output file.

If the name of a symbol matches a *pattern1* pattern that contains a wildcard character, the symbol will be renamed *pattern2* in the output file, with part of the name matching the wildcard character preserved.

Example

```
/* xxx_start will be renamed Y_start_X in the output file,
   xxx_stop will be renamed Y_stop_X in the output file. */
rename xxx_* Y_*_X
```

DIAGNOSTIC MESSAGES

This section lists the messages produced by `isymexport`:

Es001: could not open file *filename*

`isymexport` failed to open the specified file.

Es002: illegal path *pathname*

The path *pathname* is not a valid path.

Es003: format error: *message*

A problem occurred while reading the input file.

Es004: no input file

No input file was specified.

Es005: no output file

An input file, but no output file was specified.

Es006: too many input files

More than two files were specified.

Es007: input file is not an ELF executable

The input file is not an ELF executable file.

Es008: unknown directive: *directive*

The specified directive in the steering file is not recognized.

Es009: unexpected end of file

The steering file ended when more input was required.

Es010: unexpected end of line

A line in the steering file ended before the directive was complete.

Es011: unexpected text after end of directive

There is more text on the same line after the end of a steering file directive.

Es012: expected *text*

The specified text was not present in the steering file, but must be present for the directive to be correct.

Es013: pattern can contain at most one * or ?

Each pattern in the current directive can contain at most one `*` or one `?` wildcard character.

Es014: rename patterns have different wildcards

Both patterns in the current directive must contain exactly the same kind of wildcard. That is, both must either contain:

- No wildcards
- Exactly one `*`
- Exactly one `?`

This error occurs if the patterns are not the same in this regard.

Es014: ambiguous pattern match: *symbol* matches more than one rename pattern

A symbol in the input file matches more than one `rename` pattern.

Descriptions of options

This section gives detailed reference information about each command line option available for the different utilities.

--all

Syntax `--all`

Tool `ielfdumpstm8`

Description

Use this option to include the contents of all ELF sections in the output, in addition to the general properties of the input file. Sections are output in index order, except that each relocation section is output immediately after the section it holds relocations for.

By default, no section contents are included in the output.

This option is not available in the IDE.

--bin

Syntax `--bin`

Tool `ielftool`

Description

Sets the format of the output file to binary.

To set related options, choose:

Project>Options>Output converter

--checksum

Syntax

`--checksum {symbol[+offset]|address}:size,algorithm[:flags][,start];range[;range...]`

Parameters

symbol	The name of the symbol where the checksum value should be stored. Note that it must exist in the symbol table in the input ELF file.
offset	An offset to the symbol.
address	The absolute address where the checksum value should be stored.
size	The number of bytes in the checksum: `1`, `2`, or `4`; must not be larger than the size of the checksum symbol.
algorithm	The checksum algorithm used, one of: • `sum`, a byte-wise calculated arithmetic sum. The result is truncated to 8 bits. • `sum8wide`, a byte-wise calculated arithmetic sum. The result is truncated to the size of the symbol. • `sum32`, a word-wise (32 bits) calculated arithmetic sum • `crc16`, CRC16 (generating polynomial `0x11021`); used by default • `crc32`, CRC32 (generating polynomial `0x104C11DB7`) • `crc=n`, CRC with a generating polynomial of *n*.
flags	`1` specifies one's complement and `2` specifies two's complement. `m` reverses the order of the bits within each byte when calculating the checksum. For example, `2m`.
start	By default, the initial value of the checksum is 0. If necessary, use *start* to supply a different initial value.
range	The address range on which the checksum should be calculated. Hexadecimal and decimal notation is allowed (for example, `0x8002-0x8FFF`).

Tool

`ielftool`

Description

Use this option to calculate a checksum with the specified algorithm for the specified ranges. The checksum will then replace the original value in *symbol*. A new absolute

symbol will be generated; with the *symbol* name suffixed with `_value` containing the calculated checksum. This symbol can be used for accessing the checksum value later when needed, for example during debugging.

If the `--checksum` option is used more than once on the command line, the options are evaluated from left to right. If a checksum is calculated for a *symbol* that is specified in a later evaluated `--checksum` option, an error is issued.

To set related options, choose:

Project>Options>Linker>Checksum

--code

Syntax `--code`

Tool `ielfdump`

Description Use this option to dump all sections that contain executable code (sections with the ELF section attribute `SHF_EXECINSTR`).

This option is not available in the IDE.

--create

Syntax `--create libraryfile objectfile1 ... objectfileN`

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.
objectfile1 ... objectfileN	The object file(s) to build the library from.

Tool `iarchive`

Description

Use this command to build a new library from a set of object files (modules). The object files are added to the library in the exact order that they are specified on the command line.

If no command is specified on the command line, `--create` is used by default.

This option is not available in the IDE.

--delete, -d

Syntax

```
--delete libraryfile objectfile1 ... objectfileN
-d libraryfile objectfile1 ... objectfileN
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

Tool

`iarchive`

Description

Use this command to remove object files (modules) from an existing library. All object files that are specified on the command line will be removed from the library.

This option is not available in the IDE.

--edit

Syntax

```
--edit steering_file
```

Tool

`isymexport`

Description

Use this option to specify a steering file to control which symbols that are included in the `isymexport` output file, and also to rename some of the symbols if that is desired.

See also

Steering files, page 325.

This option is not available in the IDE.

--extract, -x

Syntax

```
--extract libraryfile [objectfile1 ... objectfileN]
-x libraryfile [objectfile1 ... objectfileN]
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

Tool

`iarchive`

Description

Use this command to extract object files (modules) from an existing library. If a list of object files is specified, only these files are extracted. If a list of object files is not specified, all object files in the library are extracted.

This option is not available in the IDE.

-f

Syntax

`-f filename`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.

Tool

`iarchive`, `ielfdumpstm8`, `iobjmanip`, and `isymexport`.

Description

Use this option to make the tool 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 can 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.

This option is not available in the IDE.

--fill

Syntax `--fill pattern;range[;range...]`

Parameters

range	Specifies the address range for the fill. Hexadecimal and decimal notation is allowed (for example, `0x8002–0x8FFF`). Note that each address must be 4-byte aligned.
pattern	A hexadecimal string with the `0x` prefix (for example, `0xEF`) interpreted as a sequence of bytes, where each pair of digits corresponds to one byte (for example `0x123456`, for the sequence of bytes `0x12`, `0x34`, and `0x56`). This sequence is repeated over the fill area. If the length of the fill pattern is greater than `1` byte, it is repeated as if it started at address 0.

Tool `ielftool`

Description Use this option to fill all gaps in one or more ranges with a pattern, which can be either an expression or a hexadecimal string. The contents will be calculated as if the fill pattern was repeatedly filled from the start address until the end address is passed, and then the real contents will overwrite that pattern.

If the `--fill` option is used more than once on the command line, the fill ranges cannot overlap each other.

To set related options, choose:

Project>Options>Linker>Checksum

--ihex

Syntax `--ihex`

Tool `ielftool`

Description Sets the format of the output file to linear Intel hex.

To set related options, choose:

Project>Options>Linker>Output converter

--no_strtab

Syntax `--no_strtab`

Tool `ielfdumpstm8`

Description Use this option to suppress dumping of string table sections (sections of type `SHT_STRTAB`).

This option is not available in the IDE.

--output, -o

Syntax
```
-o {filename|directory}
--output {filename|directory}
```

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 178.

Tool `iarchive` and `ielfdumpstm8`.

Description `iarchive`

By default, `iarchive` assumes that the first argument after the `iarchive` command is the name of the destination library. Use this option to explicitly specify a different filename for the library.

`ielfdumpstm8`

By default, output from the dumper is directed to the console. Use this option to direct the output to a file instead. The default name of the output file is the name of the input file with an added `id` filename extension

You can also specify the output file by specifying a file or directory following the name of the input file.

This option is not available in the IDE.

--ram_reserve_ranges

Syntax `--ram_reserve_ranges[=symbol_prefix]`

Parameters

symbol_prefix	The prefix of symbols created by this option.

Tool `isymexport`

Description Use this option to generate symbols for the areas in RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter *`symbol_prefix`*.

Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.

If `--ram_reserve_ranges` is used together with `--reserve_ranges`, the RAM areas will get their prefix from the `--ram_reserve_ranges` option and the non-RAM areas will get their prefix from the `--reserve_ranges` option.

See also *--reserve_ranges*, page 338.

This option is not available in the IDE.

--raw

Syntax `--raw`

Tool `ielfdumpstm8`

Description By default, many ELF sections will be dumped using a text format specific to a particular kind of section. Use this option to dump each selected ELF section using the generic text format.

The generic text format dumps each byte in the section in hexadecimal format, and where appropriate, as ASCII text.

This option is not available in the IDE.

--remove_section

Syntax `--remove_section {`*`section`*`|`*`number`*`}`

Parameters

`section`	The section—or sections, if there are more than one section with the same name—to be removed.
`number`	The number of the section to be removed. Section numbers can be obtained from an object dump created using `ielfdumpstm8`.

Tool iobjmanip

Description Use this option to make iobjmanip omit the specified section when generating the output file.

This option is not available in the IDE.

--rename_section

Syntax `--rename_section {oldname|oldnumber}=newname`

Parameters

oldname	The section—or sections, if there are more than one section with the same name—to be renamed.
oldnumber	The number of the section to be renamed. Section numbers can be obtained from an object dump created using ielfdumpstm8.
newname	The new name of the section.

Tool iobjmanip

Description Use this option to make iobjmanip rename the specified section when generating the output file.

This option is not available in the IDE.

--rename_symbol

Syntax `--rename_symbol oldname =newname`

Parameters

oldname	The symbol to be renamed.
newname	The new name of the symbol.

Tool iobjmanip

Description Use this option to make iobjmanip rename the specified symbol when generating the output file.

This option is not available in the IDE.

--replace, -r

Syntax
```
--replace libraryfile objectfile1 ... objectfileN
-r libraryfile objectfile1 ... objectfileN
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

Tool `iarchive`

Description

Use this command to replace or add object files (modules) to an existing library. The object files specified on the command line either replace existing object files in the library (if they have the same name) or are appended to the library.

This option is not available in the IDE.

--reserve_ranges

Syntax `--reserve_ranges[=symbol_prefix]`

Parameters

symbol_prefix	The prefix of symbols created by this option.

Tool `isymexport`

Description

Use this option to generate symbols for the areas in ROM and RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter *symbol_prefix*.

Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.

If `--reserve_ranges` is used together with `--ram_reserve_ranges`, the RAM areas will get their prefix from the `--ram_reserve_ranges` option and the non-RAM areas will get their prefix from the `--reserve_ranges` option.

See also

--ram_reserve_ranges, page 335.

This option is not available in the IDE.

--section, -s

Syntax

```
--section section_number|section_name[,...]
--s section_number|section_name[,...]
```

Parameters

section_number	The number of the section to be dumped.
section_name	The name of the section to be dumped.

Tool

`ielfdumpstm8`

Description

Use this option to dump the contents of a section with the specified number, or any section with the specified name. If a relocation section is associated with a selected section, its contents are output as well.

If you use this option, the general properties of the input file will not be included in the output.

You can specify multiple section numbers or names by separating them with commas, or by using this option more than once.

By default, no section contents are included in the output.

Example

```
-s 3,17               /* Sections #3 and #17
-s .debug_frame,42    /* Any sections named .debug_frame and
                         also section #42 */
```

This option is not available in the IDE.

--self_reloc

Syntax

`--self_reloc`

Tool

`ielftool`

Description

This option is intentionally not documented as it is not intended for general use.

This option is not available in the IDE.

--silent

Syntax
```
--silent
-S (iarchive only)
```

Tool `iarchive` and `ielftool`.

Description Causes the tool to operate without sending any messages to the standard output stream.

By default, `ielftool` sends various messages via the standard output stream. You can use this option to prevent this. `ielftool` sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

This option is not available in the IDE.

--simple

Syntax `--simple`

Tool `ielftool`

Description Sets the format of the output file to Simple code.

To set related options, choose:

Project>Options>Output converter

--srec

Syntax `--srec`

Tool `ielftool`

Description Sets the format of the output file to Motorola S-records.

To set related options, choose:

Project>Options>Output converter

--srec-len

Syntax `--srec-len=`*`length`*

Parameters

length	The number of data bytes in each S-record.

Tool `ielftool`

Description Restricts the number of data bytes in each S-record. This option can be used in combination with the `--srec` option.

This option is not available in the IDE.

--srec-s3only

Syntax `--srec-s3only`

Tool `ielftool`

Description Restricts the S-record output to contain only a subset of records, that is S3 and S7 records. This option can be used in combination with the `--srec` option.

This option is not available in the IDE.

--strip

Syntax `--strip`

Tool `iobjmanip` and `ielftool`.

Description Use this option to remove all sections containing debug information before the output file is written.

Note that `ielftool` needs an unstripped input ELF image. If you use the `--strip` option in the linker, remove it and use the `--strip` option in `ielftool` instead.

To set related options, choose:

Project>Options>Linker>Output>Include debug information in output

--symbols

Syntax

```
--symbols libraryfile
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.

Tool

`iarchive`

Description

Use this command to list all external symbols that are defined by any object file (module) in the specified library, together with the name of the object file (module) that defines it.

In silent mode (`--silent`), this command performs symbol table-related syntax checks on the library file and displays only errors and warnings.

This option is not available in the IDE.

--toc, -t

Syntax

```
--toc libraryfile
-t libraryfile
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 178.

Tool

`iarchive`

Description

Use this command to list the names of all object files (modules) in a specified library.

In silent mode (`--silent`), this command performs basic syntax checks on the library file, and displays only errors and warnings.

This option is not available in the IDE.

--verbose, -V

Syntax

```
--verbose
-V (iarchive only)
```

Tool — iarchive and ielftool.

Description — Use this option to make the tool report which operations it performs, in addition to giving diagnostic messages.

This option is not available in the IDE because this setting is always enabled.

Implementation-defined behavior

This chapter describes how IAR Systems handles the implementation-defined areas of the C language.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

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 80.

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 86.

Integer character constants with more than one character (6.4.4.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

Wide character constants with more than one character (6.4.4.4)

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Locale used for wide character constants (6.4.4.4)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Locale used for wide string literals (6.4.5)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Source characters as executive characters (6.4.5)

All source characters can be represented as executive characters.

J.3.5 Integers

Extended integer types (6.2.5)

There are no extended integer types.

Range of integer values (6.2.6.2)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types*, page 226, for information about the ranges for the different integer types.

The rank of extended integer types (6.3.1.1)

There are no extended integer types.

Signals when converting to a signed integer type (6.3.1.3)

No signal is raised when an integer is converted to a signed integer type.

Signed bitwise operations (6.5)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

J.3.6 Floating point

Accuracy of floating-point operations (5.2.4.2.2)

The accuracy of floating-point operations is unknown.

Rounding behaviors (5.2.4.2.2)

There are no non-standard values of `FLT_ROUNDS`.

Evaluation methods (5.2.4.2.2)

There are no non-standard values of `FLT_EVAL_METHOD`.

Converting integer values to floating-point values (6.3.1.4)

When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Converting floating-point values to floating-point values (6.3.1.5)

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Denoting the value of floating-point constants (6.4.4.2)

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Contraction of floating-point values (6.5)

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

Default state of FENV_ACCESS (7.6.1)

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

Additional floating-point mechanisms (7.6, 7.12)

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

Default state of FP_CONTRACT (7.12.2)

The default state of the pragma directive `FP_CONTRACT` is `OFF`.

J.3.7 Arrays and pointers

Conversion from/to pointers (6.3.2.3)

See *Casting*, page 232, for information about casting of data pointers and function pointers.

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 the pragma directive *inline*, page 255.

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 227.

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 190.

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 227.

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 225.

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 234.

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 269.

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 183.

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 169.

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 169.

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
constseg
cspy_support
dataseg
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
memory
module_name
no_pch
once
public_equ
system_include
vector
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 details, see *The DLIB runtime environment*, page 65.

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 280.

Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *fenv.h*, page 280.

Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 86.

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 89.

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 85.

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 85.

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 85.

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 90.

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 90.

clock() (7.23.2.1)

The application must supply an implementation of the `clock` function. See *Time*, page 90.

%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 90.

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 225.

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 225.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 225.

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 56: Message returned by strerror()—IAR DLIB library

A

abort
  implementation-defined behavior. 357
  system termination (DLIB) 79
absolute location
  data, placing at (@) 151
  language support for 120
  #pragma location 257
addressing. *See* memory types, data models, and code models
algorithm (STL header file) 278
alignment 225
  forcing stricter (#pragma data_alignment) 252
  of an object (__ALIGNOF__) 120
  of data types. 225
  restrictions for inline assembler 98
alignment (pragma directive) 352
__ALIGNOF__ (operator) 120
--all (ielfdump option) 329
anonymous structures 148
anonymous symbols, creating 117
application
  building, overview of 18
  startup and termination (DLIB) 77
architecture
  more information about xxvii
  of STM8. 23
argv (argument), implementation-defined behavior 346
arrays
  designated initializers in 117
  global, accessing 110
  hints about index type 147
  implementation-defined behavior. 350
  incomplete at end of structs 117
  non-lvalue 123
  of incomplete types 122
  single-value initialization. 124
asm, __asm (language extension) 118
assembler code
  calling from C 98
  calling from C++ 100
  inserting inline 97
assembler directives
  for call frame information 113
  using in inline assembler code 98
assembler instructions
  inserting inline 97
  used for calling functions. 108
assembler labels, making public (--public_equ) 201
assembler language interface 95
  calling convention. *See* assembler code
assembler list file, generating 192
assembler output file. 100
assembler, inline 118
asserts 91
  implementation-defined behavior of 354
  including in application 272
assert.h (DLIB header file) 277
atexit 91
  reserving space for calls. 58
atexit limit, setting up 58
atomic operations 38
  __monitor 245
attributes
  object 240
  type 237
auto variables 31
  at function entrance 104
  programming hints for efficient code 158
  using in inline assembler code 98
auto, packing algorithm for initializers 292

B

backtrace information *See* call frame information
Barr, Michael xxx
baseaddr (pragma directive) 353

__BASE_FILE__ (predefined symbol) 270
basic_template_matching (pragma directive) 250, 353
using 134
batch files
error return codes 171
none for building library from command line 75
--bin (ielftool option) 329
binary streams 355
bit negation 161
bitfields
data representation of 227
hints 147
implementation-defined behavior 351
non-standard types in 121
bitfields (pragma directive) 251
bits in a byte, implementation-defined behavior 347
bold style, in this guide xxxi
bool (data type) 226
adding support for in DLIB 277, 280
building_runtime (pragma directive) 353
__BUILD_NUMBER__ (predefined symbol) 270
Burrows-Wheeler algorithm, for packing initializers 292
bwt, packing algorithm for initializers 292
byte order, identifying 271

C

C and C++ linkage 102
C/C++ calling convention. *See* calling convention
C header files 277
C language, overview 117
call frame information 113
in assembler list file 99
in assembler list file (-lA) 192
call stack 113
callee-save registers, stored on stack 32
calling convention
C++, requiring C linkage 100
in compiler 101
calloc (library function) 33
See also heap
can_instantiate (pragma directive) 353
cassert (library header file) 279
cast operators
in Extended EC++ 128, 134
missing from Embedded C++ 128
casting
between pointer types 29
of pointers and integers 232
pointers to integers, language extension 123
ccomplex (library header file) 279
cctype (DLIB header file) 279
cerrno (DLIB header file) 279
cexit (system termination code)
in DLIB 77
cfenv (library header file) 279
CFI (assembler directive) 113
CFI_COMMON (call frame information macro) 116
CFI_NAMES (call frame information macro) 116
cfi.m99 (CFI header example file) 116
cfloat (DLIB header file) 279
char (data type) 226
changing default representation (--char_is_signed) . . . 183
changing representation (--char_is_unsigned) 183
implementation-defined behavior 347
signed and unsigned 227
character set, implementation-defined behavior 346
characters, implementation-defined behavior of 347
character-based I/O
in DLIB 82
overriding in runtime library 66, 74
--char_is_signed (compiler option) 183
--char_is_unsigned (compiler option) 183
checksum
calculation of 142
display format in C-SPY for symbol 146
--checksum (ielftool option) 330
cinttypes (DLIB header file) 279

ciso646 (library header file) 279
class memory (extended EC++) 130
class template partial specialization
matching (extended EC++) 133
classes 129
CLibrary (runtime model attribute) 93
climits (DLIB header file) 279
clocale (DLIB header file) 280
clock (library function), implementation-defined behavior 358
clock.c 90
__close (DLIB library function) 86
cmath (DLIB header file) 280
code
  execution of 20
  interruption of execution 37
  memory space 23
--code (ielfdump option) 331
code models 35
  calling functions in 108
  configuration 20
  identifying (__CODE_MODEL__) 270
  specifying on command line (--code_model) 183
code motion (compiler transformation) 157
  disabling (--no_code_motion) 194
codeseg (pragma directive) 353
__CODE_MODEL__ (predefined symbol) 270
__code_model (runtime model attribute) 93
--code_model (compiler option) 183
command line options
  part of compiler invocation syntax 167
  part of linker invocation syntax 167
  passing 168
  See also compiler options
  See also linker options
  typographic convention xxxi
command prompt icon, in this guide xxxi
commands, iarchive.
.comment (ELF section) 306
comments
  after preprocessor directives 123
  C++ style, using in C code 117
common block (call frame information) 113
common subexpr elimination (compiler transformation) . 156
  disabling (--no_cse) 194
compilation date
  exact time of (__TIME__) 272
  identifying (__DATE__) 270
compiler
  environment variables 169
  invocation syntax 167
  output from 170
compiler listing, generating (-l) 192
compiler object file 11
  including debug information in (--debug, -r) 185
  output from compiler 170
compiler optimization levels 154
compiler options 177
  passing to compiler 168
  reading from file (-f) 191
  specifying parameters 179
  summary 180
  syntax 177
  for creating skeleton code 99
  --warnings_affect_exit_code 171
compiler platform, identifying 271
compiler subversion number 272
compiler transformations 153
compiler version number 272
compiling
  from the command line 18
  syntax 167
complex numbers, supported in Embedded C++ 128
complex (library header file) 278
complex.h (library header file) 277
compound literals 117
computer style, typographic convention xxxi
--config (linker option) 209
--config_def (linker option) 209

configuration
basic project settings . . . 19
__low_level_init . . . 80
configuration file for linker
See also linker configuration file
configuration symbols
for file input and output . . . 85
for locale . . . 87
for printf and scanf. . . . 84
for strtod . . . 90
in library configuration files. . . . 76, 81
in linker configuration files . . . 300
specifying for linker. . . . 209
consistency, module . . . 91
const
declaring objects . . . 235
non-top level . . . 123
constseg (pragma directive) . . . 353
const_cast (cast operator) . . . 128
contents, of this guide . . . xxviii
control characters,
implementation-defined behavior . . . 359
conventions, used in this guide . . . xxx
copyright notice . . . ii
__CORE__ (predefined symbol). . . . 270
core
identifying . . . 270
specifying on command line . . . 184
__core (runtime model attribute). . . . 93
--core (compiler option) . . . 184
__cplusplus (predefined symbol) . . . 270
--cpp_init_routine (linker option) . . . 210
--create (iarchive option). . . . 331
cross call (compiler transformation) . . . 158
csetjmp (DLIB header file) . . . 280
csignal (DLIB header file) . . . 280
cspy_support (pragma directive). . . . 353
CSTACK (section) . . . 307
example . . . 139
See also stack
cstartup (system startup code)
customizing . . . 81
overriding in runtime library . . . 66, 74
cstartup.s. . . . 77
cstdarg (DLIB header file) . . . 280
cstdbool (DLIB header file) . . . 280
cstddef (DLIB header file) . . . 280
cstdio (DLIB header file) . . . 280
cstdlib (DLIB header file). . . . 280
cstring (DLIB header file). . . . 280
ctgmath (library header file) . . . 280
ctime (DLIB header file). . . . 280
ctype.h (library header file). . . . 277
cwctype.h (library header file) . . . 280
C_INCLUDE (environment variable) . . . 169
C-SPY
interface to system termination . . . 80
STL container support . . . 134
C++
See also Embedded C++ *and* Extended Embedded C++
absolute location . . . 152
calling convention . . . 100
features excluded from EC++ . . . 127
header files. . . . 278
language extensions . . . 136
special function types. . . . 41
static member variables . . . 152
support for . . . 5
C++ objects, placing in memory type . . . 31
C++ terminology. . . . xxx
C++-style comments. . . . 117
--c89 (compiler option). . . . 182

D

-D (compiler option). . . . 184
-d (iarchive option) . . . 332
--data_model (compiler option) . . . 185

data
alignment of . . . 225
different ways of storing . . . 23
located, declaring extern . . . 151
placing . . . 150, 305
at absolute location . . . 151
representation of . . . 225
storage . . . 23
data block (call frame information) . . . 113
data memory attributes, using . . . 27
data models . . . 24
configuration . . . 19
identifying (__DATA_MODEL__) . . . 270
large . . . 25
medium . . . 25
small . . . 25
data pointers . . . 232
data types . . . 226
avoiding signed . . . 147
avoiding 32-bit . . . 147
floating point . . . 230
in C++ . . . 236
integers . . . 226
dataseg (pragma directive) . . . 353
data_alignment (pragma directive) . . . 252
__DATA_MODEL__ (predefined symbol) . . . 270
__data_model (runtime model attribute) . . . 93
__DATE__ (predefined symbol) . . . 270
date (library function), configuring support for . . . 90
DC32 (assembler directive) . . . 98
--debug (compiler option) . . . 185
debug information, including in object file . . . 185
.debug (ELF section) . . . 306
--debug_lib (linker option) . . . 210
decimal point, implementation-defined behavior . . . 359
declarations
empty . . . 124
in for loops . . . 117
Kernighan & Ritchie . . . 160
of functions . . . 102
declarations and statements, mixing . . . 117
define block (linker directive) . . . 289
define overlay (linker directive) . . . 290
define symbol (linker directive) . . . 300
--define_symbol (linker option) . . . 210
define_type_info (pragma directive) . . . 353
--delete (iarchive option) . . . 332
delete (keyword) . . . 33
denormalized numbers. *See* subnormal numbers
--dependencies (compiler option) . . . 185
--dependencies (linker option) . . . 211
deque (STL header file) . . . 279
destructors and interrupts, using . . . 135
device description files, preconfigured . . . 6
diagnostic messages . . . 172
classifying as compilation errors . . . 186
classifying as compilation remarks . . . 187
classifying as compiler warnings . . . 188
classifying as linker warnings . . . 213
classifying as linking errors . . . 211
classifying as linking remarks . . . 212
disabling compiler warnings . . . 198
disabling linker warnings . . . 219
disabling wrapping of in compiler . . . 198
disabling wrapping of in linker . . . 220
enabling compiler remarks . . . 202
enabling linker remarks . . . 221
listing all used by compiler . . . 188
listing all used by linker . . . 213
suppressing in compiler . . . 187
suppressing in linker . . . 212
diagnostics
iarchive . . . 317
iobjmanip . . . 322
isymexport . . . 327
--diagnostics_tables (compiler option) . . . 188
--diagnostics_tables (linker option) . . . 213
diagnostics, implementation-defined behavior . . . 345

diag_default (pragma directive) 252
--diag_error (compiler option) 186
--diag_error (linker option) 211
diag_error (pragma directive) 252
--diag_remark (compiler option). 187
--diag_remark (linker option) 212
diag_remark (pragma directive) 253
--diag_suppress (compiler option) 187
--diag_suppress (linker option) 212
diag_suppress (pragma directive) 253
--diag_warning (compiler option) 188
--diag_warning (linker option) 213
diag_warning (pragma directive) 253
DIFUNCT (section) 307
directives
  pragma 7, 249
  to the linker 283
directory, specifying as parameter. 178
__disable_interrupt (intrinsic function). 266
--discard_unused_publics (compiler option). 188
disclaimer ii
DLIB. 21, 276
  configurations 81
  configuring. 66, 189
  including debug support. 71
  reference information. *See* the online help system 275
  runtime environment 65
--dlib_config (compiler option). 189
DLib_Defaults.h (library configuration file) 76, 81
__DLIB_FILE_DESCRIPTOR (configuration symbol) . . . 85
dlstm8*libname*.h 76
do not initialize (linker directive) 294
document conventions. xxx
documentation, library 275
domain errors, implementation-defined behavior 354
double (data type) 230
do_not_instantiate (pragma directive). 353
dynamic initialization 77
dynamic initialization for C++ 50
dynamic memory 33

E

-e (compiler option) 190
early_initialization (pragma directive) 353
--ec++ (compiler option). 190
EC++ header files 278
--edit (isymexport option) 332
edition, of this guide ii
--eec++ (compiler option). 190
__eeprom (extended keyword) 242
eeprom (memory type) 27
ELF utilities 315
Embedded C++. 127
  differences from C++. 127
  enabling 190
  function linkage 102
  language extensions 127
  overview 127
Embedded C++ Technical Committee xxx
embedded systems, IAR special support for 6
__embedded_cplusplus (predefined symbol) 271
empty region (in linker configuration file) 288
__enable_interrupt (intrinsic function) 266
--enable_multibytes (compiler option) 190
--entry (linker option) 213
entry label, program 77
enumerations, implementation-defined behavior. 351
enums
  data representation 227
  forward declarations of 122
environment
  implementation-defined behavior. 346
  runtime (DLIB) 65
environment names, implementation-defined behavior . . . 347
environment variables
  C_INCLUDE. 169
  ILINKSTM8_CMD_LINE 169

QCCSTM8 . . . 169
environment (native), implementation-defined behavior . . . 359–360
EQU (assembler directive) . . . 201
ERANGE . . . 354
errno value at underflow, implementation-defined behavior . . . 357
errno.h (library header file) . . . 277
error messages . . . 174
classifying for compiler . . . 186
classifying for linker . . . 211
range . . . 62
error return codes . . . 171
error (pragma directive) . . . 254
--error_limit (compiler option) . . . 191
--error_limit (linker option) . . . 214
escape sequences, implementation-defined behavior . . . 347
exception handling, missing from Embedded C++ . . . 127
exception (library header file) . . . 278
_Exit (library function) . . . 79
exit (library function) . . . 79
implementation-defined behavior . . . 357
_exit (library function) . . . 79
__exit (library function) . . . 79
export keyword, missing from Extended EC++ . . . 132
export (linker directive) . . . 301
--export_builtin_config (linker option) . . . 214
expressions (in linker configuration file) . . . 301
extended command line file
for compiler . . . 191
for linker . . . 214
passing options . . . 168
Extended Embedded C++ . . . 128
enabling . . . 190
standard template library (STL) . . . 278
extended keywords . . . 237
enabling (-e) . . . 190
overview . . . 7
summary . . . 241
syntax . . . 28
object attributes . . . 240
type attributes on data objects . . . 238
type attributes on data pointers . . . 239
type attributes on function pointers . . . 240
type attributes on functions . . . 239
extended-selectors (in linker configuration file) . . . 299
extern "C" linkage . . . 132
--extract (iarchive option) . . . 333

F

-f (compiler option) . . . 191
-f (iobjmanip option) . . . 333
-f (linker option) . . . 214
__far (extended keyword) . . . 242
far (memory type) . . . 27
__far_func (extended keyword) . . . 243
__far_func (function pointer) . . . 231
fatal error messages . . . 174
fdopen, in stdio.h . . . 281
fegettrapdisable . . . 280
fegettrapenable . . . 280
FENV_ACCESS, implementation-defined behavior . . . 350
fenv.h (library header file) . . . 277, 279
additional C functionality . . . 280
fgetpos (library function), implementation-defined behavior . . . 357
__FILE__ (predefined symbol) . . . 271
file buffering, implementation-defined behavior . . . 355
file dependencies, tracking . . . 185
file paths, specifying for #include files . . . 192
file position, implementation-defined behavior . . . 355
file (zero-length), implementation-defined behavior . . . 355
filename
extension for device description files . . . 6
extension for header files . . . 6
extension for linker configuration files . . . 6
of object executable image . . . 220
search procedure for . . . 169

specifying as parameter 178
filenames (legal), implementation-defined behavior 355
fileno, in stdio.h 281
files, implementation-defined behavior
handling of temporary 356
multibyte characters in. 356
opening 356
--fill (ielftool option). 334
float (data type). 230
floating-point constants, hexadecimal notation 117
floating-point environment
accessing or not 262
floating-point expressions, contracting or not 263
floating-point format. 230
hints 148
implementation-defined behavior. 349
special cases. 231
32-bits 230
floating-point status flags 280
float.h (library header file) 277
FLT_EVAL_METHOD, implementation-defined behavior 349, 354, 358
FLT_ROUNDS, implementation-defined behavior 349, 358
for loops, declarations in. 117
--force_output (linker option) 215
formats
floating-point values 230
standard IEEE (floating point) 230
FP_CONTRACT, implementation-defined behavior. 350
fragmentation, of heap memory 33
free (library function). *See also* heap 33
fsetpos (library function), implementation-defined behavior 357
fstream (library header file) 278
ftell (library function), implementation-defined behavior . 357
Full DLIB (library configuration) 81
__func__ (predefined symbol) 124, 271
__FUNCTION__ (predefined symbol) 124, 271
function calls
calling convention 101
large code model 109
medium code model. 109
small code model 108
stack image after 105
function declarations, Kernighan & Ritchie 160
function inlining (compiler transformation) 156
disabling (--no_inline) 195
function pointers 231
function prototypes 160
enforcing 202
function return addresses 106
function template parameter deduction (extended EC++) . 133
function (pragma directive). 353
functional (STL header file) 279
functions 35
calling in different code models 108
C++ and special function types 41
declaring 102, 160
inlining. 117, 156, 158, 255
interrupt 37–38
intrinsic 95, 159
monitor 38
parameters 104
placing in memory 150, 152
recursive
avoiding 160
storing data on stack 32
reentrancy (DLIB) 276
related extensions. 35
return values from 106
special function types. 37

G

getenv (library function), configuring support for 88
getw, in stdio.h 281
getzone (library function), configuring support for 90

getzone.c . . . 90
__get_interrupt_state (intrinsic function) . . . 266
global arrays, accessing . . . 110
global variables, accessing . . . 110
GRP_COMDAT, group type . . . 323
guidelines, reading . . . xxvii

H

__halt (intrinsic function) . . . 267
HALT (assembler instruction) . . . 267
Harbison, Samuel P. . . . xxx
hardware support in compiler . . . 65
hash_map (STL header file) . . . 279
hash_set (STL header file) . . . 279
hdrstop (pragma directive) . . . 353
header files
  C . . . 277
  C++ . . . 278
  EC++ . . . 278
  library . . . 275
  special function registers . . . 162
  STL . . . 278
  DLib_Defaults.h . . . 76, 81
  dlstm8*libname*.h . . . 76
  intrinsics.h . . . 265
  stdbool.h . . . 226, 277
  stddef.h . . . 227
header names, implementation-defined behavior . . . 352
--header_context (compiler option) . . . 192
heap
  dynamic memory . . . 33
  storing data . . . 24
heap size
  and standard I/O . . . 140
  changing default . . . 58
HEAP (section) . . . 140, 309
heap (zero-sized), implementation-defined behavior . . . 357
hide (isymexport directive) . . . 326
hints
  for good code generation . . . 158
  implementation-defined behavior . . . 350
  using efficient data types . . . 147
__huge (extended keyword) . . . 243
huge (memory type) . . . 27
__huge_func (extended keyword) . . . 244
__huge_func (function pointer) . . . 232

I

-I (compiler option) . . . 192
IAR Command Line Build Utility . . . 75
IAR Systems Technical Support . . . 175
iarbuild.exe (utility) . . . 75
iarchive . . . 315
  commands summary . . . 316
  options summary . . . 317
__IAR_SYSTEMS_ICC__ (predefined symbol) . . . 271
.iar.debug (ELF section) . . . 306
.iar.dynexit (section) . . . 310
__ICCSTM8__ (predefined symbol) . . . 271
icons, in this guide . . . xxxi
IDE
  building a library from . . . 75
  overview of build tools . . . 3
identifiers, implementation-defined behavior . . . 347
IEEE format, floating-point values . . . 230
ielfdump . . . 320
  options summary . . . 321
ielftool . . . 318
  options summary . . . 319
if (linker directive) . . . 303
--ihex (ielftool option) . . . 334
ILINK
  See linker
ILINK options. *See* linker options
ILINKSTM8_CMD_LINE (environment variable) . . . 169
--image_input (linker option) . . . 215

implementation-defined behavior 345
important_typedef (pragma directive) 353
include files
  including before source files 200
  specifying 169
include (linker directive) 304
include_alias (pragma directive) 254
infinity 231
infinity (style for printing), implementation-defined behavior 356
inheritance, in Embedded C++ 127
initialization
  changing default 58
  C++ dynamic 50
  dynamic 77
  manual 59
  packing algorithm for 58
  single-value 124
initialize (linker directive) 291
initializers, static 123
inline assembler 97, 118
  avoiding 159
  See also assembler language interface
inline functions 117
  in compiler 156
inline (pragma directive) 255
inlining functions, implementation-defined behavior 351
instantiate (pragma directive) 353
integers 226
  casting 232
  implementation-defined behavior 349
  intptr_t 233
  ptrdiff_t 233
  size_t 232
  uintptr_t 233
integral promotion 161
Intel hex 139
internal error 174
__interrupt (extended keyword) 38, 244
  using in pragma directives 263
interrupt functions 37
interrupt state, restoring 267
interrupt vector, specifying with pragma directive 263
interrupts
  disabling 245
    during function execution 38
  processor state 31
  using with EC++ destructors 135
intptr_t (integer type) 233
__intrinsic (extended keyword) 244
intrinsic functions 159
  overview 95
  summary 265
intrinsics.h (header file) 265
introduction
  linker configuration file 283
  linking 43
inttypes.h (library header file) 277
.intvec (section) 311
invocation syntax 167
iobjmanip 321
  options summary 322
iomanip (library header file) 278
ios (library header file) 278
iosfwd (library header file) 278
iostream (library header file) 278
ISO/ANSI C
  C++ features excluded from EC++ 127
  specifying strict usage 203
iso646.h (library header file) 277
istream (library header file) 278
isymexport 324
  options summary 325
italic style, in this guide xxxi
iterator (STL header file) 279
I/O module, overriding in runtime library 66, 74

K

--keep (linker option) 216
keep (linker directive) 294
keep_definition (pragma directive) 353
Kernighan & Ritchie function declarations 160
    disallowing 202
Kernighan, Brian W. xxx
keywords 237
    extended, overview of 7

L

-l (compiler option) 192
    for creating skeleton code 99
labels 123
    assembler, making public 201
    __program_start 77
Labrosse, Jean J. xxx
Lajoie, Josée xxx
language extensions
    Embedded C++ 127
    enabling 256
    enabling (-e) 190
language overview 5
language (pragma directive) 256
Large code model, function calls 109
large (code model) 36
LDF (assembler instruction) 29, 111
Lempel-Ziv-Welch algorithm, for packing initializers . . . 292
libraries
    runtime 67
    standard template library 278
library configuration files
    DLIB 81
    DLib_Defaults.h 76, 81
    dlstm8*libname*.h 76
    modifying 76
    specifying 189
library documentation 275
library features, missing from Embedded C++ 128
library functions 275
    reference information xxix
    summary, DLIB 277
library header files 275
library modules
    introduction 43
    overriding 74
library object files 275
library project template 21
    using 75
lightbulb icon, in this guide xxxi
limits.h (library header file) 277
__LINE__ (predefined symbol) 271
linkage, C and C++ 102
linker 43
    output from 172
linker configuration file
    for placing code and data 46
    in depth 283
    selecting 53
linker object executable image
    specifying filename of (-o) 220
linker options 207
    reading from file (-f) 214
    summary 207
    typographic convention xxxi
linking
    from the command line 18
    process for 44
    process for, an overview 12
Lippman, Stanley B. xxx
list (STL header file) 279
listing, generating 192
literals, compound 117
literature, recommended xxx
__LITTLE_ENDIAN__ (predefined symbol) 271
local variables, *See* auto variables

locale support
  DLIB . . . 86
    adding . . . 88
    changing at runtime. . . . 88
    removing . . . 87
locale, implementation-defined behavior . . . 348, 359
locale.h (library header file) . . . 277
located data, declaring extern . . . 151
location (pragma directive) . . . 151, 257
--log (linker option) . . . 216
--log_file (linker option) . . . 216
long double (data type) . . . 230
long float (data type), synonym for double . . . 123
loop overhead, reducing . . . 197
loop unrolling (compiler transformation) . . . 156
  disabling . . . 197
loop-invariant expressions. . . . 157
__low_level_init . . . 77
  customizing . . . 80
low-level processor operations . . . 119, 265
  accessing . . . 95
__lseek (library function) . . . 86
lzw, packing algorithm for initializers. . . . 292

M

-map (linker option) . . . 217
macros
  embedded in #pragma optimize . . . 258
  ERANGE (in errno.h) . . . 354
  inclusion of assert . . . 272
  NULL, implementation-defined behavior . . . 355
  substituted in #pragma directives . . . 119
  variadic . . . 117
main (function), implementation-defined behavior . . . 346
malloc (library function)
  See also heap . . . 33
--mangled_names_in_messages (linker option) . . . 217
Mann, Bernhard . . . xxx
map file, producing . . . 217
map (STL header file) . . . 279
math functions rounding mode, implementation-defined behavior . . . 358
math.h (library header file) . . . 277
MB_LEN_MAX, implementation-defined behavior . . . 358
Medium code model, function calls . . . 109
medium (code model) . . . 36
member functions, pointers to . . . 135
memory
  accessing . . . 19, 26, 109
    using far method . . . 111–112
    using huge method . . . 111
    using near method . . . 111
    using tiny method . . . 110
  allocating in C++ . . . 33
  dynamic . . . 33
  heap . . . 33
  non-initialized . . . 163
  RAM, saving . . . 160
  releasing in C++. . . . 33
  stack. . . . 31
    saving . . . 160
  used by global or static variables . . . 24
memory layout, STM8 . . . 23
memory management, type-safe . . . 127
memory map, output from linker . . . 172
memory placement
  using pragma directive . . . 28
  using type definitions. . . . 28, 239
memory types . . . 26
  C++ . . . 31
  placing variables in . . . 31
  pointers . . . 29
  specifying . . . 27
  structures . . . 30
  summary . . . 27
  using for efficient coding . . . 159
memory (pragma directive). . . . 353
memory (STL header file). . . . 279

__memory_of, operator. 131
message (pragma directive). 257
messages
disabling 202, 222
forcing 257
--mfc (compiler option). 193
--misrac_verbose (compiler option) 181
--misrac_verbose (linker option). 208
--misrac1998 (compiler option) 181
--misrac1998 (linker option) 208
--misrac2004 (compiler option) 181
--misrac2004 (linker option) 208
mode changing, implementation-defined behavior 356
module consistency. 91
rtmodel. 260
modules, introduction 43
module_name (pragma directive) 353
__monitor (extended keyword) 245
monitor functions 38, 244
Motorola S-records 139
multibyte character support. 190
multibyte characters, implementation-defined
behavior 347, 359
multiple inheritance
in Extended EC++ 128
missing from Embedded C++ 127
missing from STL 128
multi-file compilation 154
mutable attribute, in Extended EC++ 128, 135

N

names block (call frame information) 113
namespace support
in Extended EC++ 128, 135
missing from Embedded C++ 128
naming conventions xxxi
NaN, implementation-defined behavior 356
native environment,
implementation-defined behavior 359–360
NDEBUG (preprocessor symbol) 272
__near (extended keyword). 245
near (memory type). 26
__near_func (extended keyword) 245
__near_func (function pointer) 231
new (keyword) 33
new (library header file) 278
non-initialized variables, hints for. 163
non-scalar parameters, avoiding 160
NOP (assembler instruction). 267
__noreturn (extended keyword) 246
Normal DLIB (library configuration) 81
Not a number (NaN) 231
--no_code_motion (compiler option) 194
--no_cse (compiler option) 194
--no_fragments (compiler option). 195
--no_fragments (linker option) 218
__no_init (extended keyword) 163, 246
--no_inline (compiler option) 195
--no_library_search (linker option). 218
--no_locals (linker option) 219
__no_operation (intrinsic function). 267
--no_path_in_file_macros (compiler option). 195
no_pch (pragma directive) 353
--no_range_reservations (linker option) 219
--no_remove (linker option) 219
--no_static_destruction (compiler option). 196
--no_system_include (compiler option) 196
--no_typedefs_in_diagnostics (compiler option). 197
--no_unroll (compiler option) 197
--no_warnings (compiler option) 198
--no_warnings (linker option) 219
--no_wrap_diagnostics (compiler option) 198
--no_wrap_diagnostics (linker option) 220
NULL, implementation-defined behavior 355
numbers (in linker configuration file) 302
numeric (STL header file). 279

O

-O (compiler option) 198
-o (iarchive option) 335
-o (ielfdump option) 335
-o (linker option) 220
object attributes 240
object filename
  specifying 335
object_attribute (pragma directive) 163, 258
once (pragma directive) 353
--only_stdout (compiler option) 199
--only_stdout (linker option) 220
__open (library function) 86
operators
  See also @ (operator)
  __memory_of 131
optimization
  code motion, disabling 194
  common sub-expression elimination, disabling 194
  configuration 20
  disabling 155
  function inlining, disabling (--no_inline) 195
  hints 158
  loop unrolling, disabling 197
  specifying (-O) 198
  techniques 155
  type-based alias analysis, disabling (--tbaa) 196
  using inline assembler code 98
  using pragma directive 258
optimization levels 154
optimize (pragma directive) 258
option parameters 177
options, compiler. *See* compiler options
options, iarchive. *See* iarchive options
options, ielfdump. *See* ielfdump options
options, ielftool. *See* ielftool options
options, iobjmanip. *See* iobjmanip options
options, isymexport. *See* isymexport options
options, linker. *See* linker options
Oram, Andy xxx
ostream (library header file) 278
output
  from preprocessor 200
  specifying for linker 18
--output (compiler option) 199
--output (iarchive option) 335
--output (ielfdump option) 335
--output (linker option) 220
overhead, reducing 156

P

packbits, packing algorithm for initializers 292
packing, algorithms for initializers 292
parameters
  function 104
  hidden 104
  non-scalar, avoiding 160
  register 104–105
  rules for specifying a file or directory 178
  specifying 179
  stack 104–105
  typographic convention xxxi
part number, of this guide ii
permanent registers 104
place at (linker directive) 295
place in (linker directive) 296
placement
  code and data 305
  in named sections 152
  of code and data, introduction to 46
--place_holder (linker option) 220
plain char, implementation-defined behavior 347
pointer types 231
  differences between 29
  mixing 123
  using the best 148

pointers
casting . . . 29, 232
data . . . 232
function . . . 231
implementation-defined behavior. . . 350
polymorphism, in Embedded C++ . . . 127
pow (library function). . . 90
powXp (library function) . . . 90
pragma directives . . . 7
summary . . . 249
basic_template_matching, using . . . 134
for absolute located data . . . 151
list of all recognized. . . 352
type_attribute, using. . . 28
_Pragma (preprocessor operator) . . . 117
predefined symbols
overview . . . 7
summary . . . 270
--predef_macro (compiler option). . . 200
--preinclude (compiler option) . . . 200
--preprocess (compiler option) . . . 200
preprocessor
output. . . 200
overview of . . . 269
preprocessor directives
comments at the end of . . . 123
implementation-defined behavior. . . 352
#pragma . . . 249
preprocessor extensions
__VA_ARGS__ . . . 117
#warning message . . . 273
preprocessor operator, _Pragma() . . . 117
preprocessor symbols . . . 270
defining . . . 184, 210
preserved registers . . . 104
__PRETTY_FUNCTION__ (predefined symbol). . . 271
primitives, for special functions . . . 37
print formatter, selecting . . . 70
printf (library function). . . 69
choosing formatter . . . 69
configuration symbols . . . 84
implementation-defined behavior. . . 356
__printf_args (pragma directive). . . 259
printing characters, implementation-defined behavior . . . 359
processor configuration. . . 19
processor operations
accessing . . . 95
low-level . . . 119, 265
program entry label. . . 77
program termination, implementation-defined behavior . . 346
programming hints . . . 158
__program_start (label). . . 77
projects
basic settings for . . . 19
setting up for a library . . . 75
prototypes, enforcing . . . 202
ptrdiff_t (integer type). . . 233
PUBLIC (assembler directive) . . . 201
publication date, of this guide . . . ii
--public_equ (compiler option) . . . 201
public_equ (pragma directive) . . . 353
putenv (library function), absent from DLIB . . . 88
putw, in stdio.h . . . 281

Q

QCCSTM8 (environment variable). . . 169
qualifiers
const and volatile . . . 234
implementation-defined behavior. . . 351
queue (STL header file) . . . 279

R

-r (iarchive option) . . . 338
raise (library function), configuring support for . . . 89
raise.c . . . 89

RAM
example of declaring region . . . 47
initializers copied from ROM . . . 17
running code from . . . 60
saving memory . . . 160
--ram_reserve_ranges (isymexport option) . . . 335
range errors . . . 62
--raw (ielfdump option) . . . 336
__read (library function) . . . 86
customizing . . . 82
read formatter, selecting . . . 71
reading guidelines . . . xxvii
reading, recommended . . . xxx
realloc (library function)
See also heap . . . 33
recursive functions
avoiding . . . 160
storing data on stack . . . 32
--redirect (linker option) . . . 221
reentrancy (DLIB) . . . 276
reference information, typographic convention . . . xxxi
region . . . 47
region expression (in linker configuration file) . . . 287
region literal (in linker configuration file) . . . 285
register keyword, implementation-defined behavior . . . 350
register parameters . . . 104–105
registered trademarks . . . ii
registers
assigning to parameters . . . 105
callee-save, stored on stack . . . 32
for function returns . . . 106
in assembler-level routines . . . 101
preserved . . . 104
scratch . . . 104
virtual . . . 102
reinterpret_cast (cast operator) . . . 128
.rel (ELF section) . . . 306
.rela (ELF section) . . . 306
--relaxed_fp (compiler option) . . . 201
relocation errors, resolving . . . 62
remark (diagnostic message) . . . 174
classifying for compiler . . . 187
classifying for linker . . . 212
enabling in compiler . . . 202
enabling in linker . . . 221
--remarks (compiler option) . . . 202
--remarks (linker option) . . . 221
remove (library function) . . . 86
implementation-defined behavior . . . 356
--remove_section (iobjmanip option) . . . 336
remquo, magnitude of . . . 354
rename (isymexport directive) . . . 327
rename (library function) . . . 86
implementation-defined behavior . . . 356
--rename_section (iobjmanip option) . . . 337
--rename_symbol (iobjmanip option) . . . 337
--replace (iarchive option) . . . 338
__ReportAssert (library function) . . . 91
required (pragma directive) . . . 259
--require_prototypes (compiler option) . . . 202
--reserve_ranges (isymexport option) . . . 338
return addresses . . . 106
return values, from functions . . . 106
RIM (assembler instruction) . . . 266
Ritchie, Dennis M. . . . xxx
__root (extended keyword) . . . 246
routines, time-critical . . . 95, 119, 265
rtmodel (assembler directive) . . . 92
rtmodel (pragma directive) . . . 260
rtti support, missing from STL . . . 128
__rt_version (runtime model attribute) . . . 93
runtime environment, DLIB . . . 65
runtime libraries
choosing . . . 21
introduction . . . 275
DLIB . . . 67
choosing . . . 67
customizing without rebuilding . . . 69

naming convention 68
overriding modules in 66, 74
runtime model attributes 91
runtime model definitions 260
runtime type information, missing from Embedded C++ . 128

S

-S (iarchive option) 340
-s (ielfdump option) 339
scanf (library function)
choosing formatter 70
configuration symbols 84
implementation-defined behavior. 357
__scanf_args (pragma directive) 261
scratch registers 104
--section (ielfdump option) 339
section names, declaring 261
section (pragma directive). 261
sections 305
summary 305
introduction 43
__section_begin (extended operator) 121
__section_end (extended operator) 121
__section_size (extended operator) 121
section-selectors (in linker configuration file). 297
section, allocation of 46
segment (pragma directive) 261
semaphores
C example 38
C++ example 40
operations on 245
set (STL header file) 279
setjmp.h (library header file). 277
setlocale (library function) 88
settings, basic for project configuration 19
__set_interrupt_state (intrinsic function) 267
severity level, of diagnostic messages 174
specifying 174
SFR
accessing special function registers 162
declaring extern special function registers 151
shared object. 171, 217
short (data type) 226
show (isymexport directive) 326
.shstrtab (ELF section) 306
signal (library function)
configuring support for 89
implementation-defined behavior. 354
signals, implementation-defined behavior. 346
at system startup 346
signal.c 89
signal.h (library header file) 277
signed char (data type) 226–227
specifying 183
signed int (data type). 226
signed long long (data type) 226
signed long (data type) 226
signed short (data type). 226
signed values, avoiding 147
--silent (compiler option) 202
--silent (iarchive option) 340
--silent (ielftool option). 340
--silent (linker option). 222
silent operation
specifying in compiler 202
specifying in linker 222
SIM (assembler instruction) 266
--simple (ielftool option). 340
size_t (integer type) 232
skeleton code, creating for assembler language interface . . 98
skeleton.s (assembler source output). 99
slist (STL header file) 279
small (code model) 36
smallest, packing algorithm for initializers 292
source files, list all referred. 192
space characters, implementation-defined behavior 355
special function registers (SFR) 162

special function types 37
overview 7
sprintf (library function) 69
choosing formatter 69
--srec (ielftool option) 340
--srec-len (ielftool option) 341
--srec-s3only (ielftool option) 341
sscanf (library function), choosing formatter 70
sstream (library header file) 278
stack 31
advantages and problems using 32
cleaning after function return 106
contents of 31
internal data 307
layout 105
saving space 160
size 139
stack parameters 104–105
stack pointer 32
stack pointer register, considerations 104
stack (STL header file) 279
standard error
redirecting in compiler 199
redirecting in linker 220
standard input 82
standard output 82
specifying in compiler 199
specifying in linker 220
standard template library (STL)
in Extended EC++ 128, 134, 278
missing from Embedded C++ 128
startup system. *See* system startup
static variables 24
taking the address of 158
static_cast (cast operator) 128
status flags for floating-point 280
std namespace, missing from EC++
and Extended EC++ 135
stdarg.h (library header file) 277
stdbool.h (library header file) 226, 277
__STDC__ (predefined symbol) 272
STDC CX_LIMITED_RANGE (pragma directive) 262
STDC FENV_ACCESS (pragma directive) 262
STDC FP_CONTRACT (pragma directive) 262
__STDC_VERSION__ (predefined symbol) 272
stddef.h (library header file) 227, 277
stderr 86, 199, 220
stdexcept (library header file) 278
stdin 86
stdint.h (library header file) 277, 280
stdio.h (library header file) 277
stdio.h, additional C functionality 281
stdlib.h (library header file) 277
stdout 86, 199, 220
implementation-defined behavior 355
Steele, Guy L. xxx
steering file, input to isymexport 325
STL 134
STM8
instruction set 109
memory access 19
memory layout 23
supported devices 6
strcasecmp, in string.h 281
strdup, in string.h 281
streambuf (library header file) 278
streams, implementation-defined behavior 346
streams, supported in Embedded C++ 128
strerror (library function), implementation-defined
behavior 360
--strict (compiler option) 203
string (library header file) 278
strings, supported in Embedded C++ 128
string.h (library header file) 277
string.h, additional C functionality 281
--strip (ielftool option) 341
--strip (iobjmanip option) 341
--strip (linker option) 222
strncasecmp, in string.h 281
strnlen, in string.h 281

Stroustrup, Bjarne . . . xxx
strstream (library header file) . . . 278
--strtab (ielfdump option) . . . 335, 339
.strtab (ELF section) . . . 306
strtod (library function), configuring support for . . . 90
structure types, layout of . . . 233
structures
  accessing using a pointer . . . 110
  anonymous . . . 120, 148
  implementation-defined behavior . . . 351
  placing in memory type . . . 30
subnormal numbers . . . 231
__SUBVERSION__ (predefined symbol) . . . 272
support, technical . . . 175
symbols
  anonymous, creating . . . 117
  directing from one to another . . . 221
  including in output . . . 259
  overview of predefined . . . 7
  preprocessor, defining . . . 184, 210
--symbols (iarchive option) . . . 342
.symtab (ELF section) . . . 306
syntax
  command line options . . . 177
  extended keywords . . . 28, 238–240
  invoking compiler and linker . . . 167
system function, implementation-defined behavior . . 347, 357
system startup
  customizing . . . 80
  DLIB . . . 77
  initialization phase . . . 14
system termination
  C-SPY interface to . . . 80
  DLIB . . . 79
system (library function), configuring support for . . . 88
system_include (pragma directive) . . . 353
--system_include_dir (compiler option) . . . 203

T

-t (iarchive option) . . . 342
__task (extended keyword) . . . 247
technical support, IAR Systems . . . 175
template support
  in Extended EC++ . . . 128, 132
  missing from Embedded C++ . . . 127
Terminal I/O window
  making available (DLIB) . . . 73
  not supported when . . . 74
terminal I/O, debugger runtime interface for . . . 72
terminal output, speeding up . . . 73
termination of system. *See* system termination
termination status, implementation-defined behavior . . . 357
terminology . . . xxx
tgmath.h (library header file) . . . 277
32-bits (floating-point format) . . . 230
this (pointer) . . . 100
  class memory . . . 130
  referring to a class object . . . 130
__TIME__ (predefined symbol) . . . 272
time zone (library function), implementation-defined behavior . . . 357
time (library function), configuring support for . . . 90
time-critical routines . . . 95, 119, 265
time.c . . . 90
time.h (library header file) . . . 277
tiny (memory type) . . . 26
__tiny (extended keyword) . . . 247
tips, programming . . . 158
--toc (iarchive option) . . . 342
tools icon, in this guide . . . xxxi
trademarks . . . ii
transformations, compiler . . . 153
translation, implementation-defined behavior . . . 345
__trap (intrinsic function) . . . 267
trap vectors, specifying with pragma directive . . . 263
TRAP (assembler instruction) . . . 267

type attributes 237
specifying 263
type definitions, used for specifying memory storage . 28, 239
type qualifiers
const and volatile 234
implementation-defined behavior. 351
typedefs
excluding from diagnostics 197
repeated 123
type_attribute (pragma directive) 28, 263
type-based alias analysis (compiler transformation) 157
disabling 196
type-safe memory management 127
typographic conventions xxxi

U

uchar.h (library header file). 277
uintptr_t (integer type) 233
underflow errors, implementation-defined behavior 354
unions
anonymous. 120, 148
implementation-defined behavior. 351
universal character names, implementation-defined behavior 352
unsigned char (data type) 226–227
changing to signed char 183
unsigned int (data type). 226
unsigned long long (data type) 226
unsigned long (data type) 226
unsigned short (data type). 226
--use_unix_directory_separators (compiler option). 203
utilities (ELF) 315
utility (STL header file) 279

V

-V (iarchive option). 342
variables
auto 31
defined inside a function 31
global
accessing. 110
placement in memory 24
hints for choosing 158
local. *See* auto variables
non-initialized 163
placing at absolute addresses 152
placing in named sections 152
static
placement in memory 24
taking the address of 158
vector (pragma directive) 38, 263, 353
vector (STL header file) 279
__VER__ (predefined symbol) 272
--verbose (iarchive option) 342
--verbose (ielftool option) 342
version, IAR Embedded Workbench ii
version, of compiler 272
virtual registers 102
--vla (compiler option) 204
void, pointers to 122
volatile (keyword). 161
volatile, declaring objects 234–235
--vregs (compiler option) 204

W

__wait_for_interrupt (intrinsic function). 267
#warning message (preprocessor extension) 273
warnings 174
classifying in compiler. 188
classifying in linker 213
disabling in compiler 198
disabling in linker 219
exit code in compiler 204
exit code in linker 222

warnings icon, in this guide xxxi
warnings (pragma directive) 353
--warnings_affect_exit_code (compiler option) 171, 204
--warnings_affect_exit_code (linker option) 222
--warnings_are_errors (compiler option) 205
--warnings_are_errors (linker option) 222
wchar_t (data type), adding support for in C. 227
wchar.h (library header file) 278, 280
wctype.h (library header file) 278
__weak (extended keyword) 248
weak (pragma directive) 264
web sites, recommended xxx
WFE (assembler instruction) 267
WFI (assembler instruction) 267
white-space characters, implementation-defined behavior 345
__write (library function) 86
  customizing 82

X

-x (iarchive option) 333
xreportassert.c. 91

Z

zeros, packing algorithm for initializers 292

Symbols

_Exit (library function) 79
_exit (library function) 79
__ALIGNOF__ (operator) 120
__asm (language extension) 118
__BASE_FILE__ (predefined symbol) 270
__BUILD_NUMBER__ (predefined symbol) 270
__close (library function) 86
__code_model (runtime model attribute) 93
__CODE_MODEL__ (predefined symbol). 270
__core (runtime model attribute). 93
__CORE__ (predefined symbol). 270
__cplusplus (predefined symbol) 270
__data_model (runtime model attribute). 93
__DATA_MODEL__ (predefined symbol) 270
__DATE__ (predefined symbol) 270
__disable_interrupt (intrinsic function). 266
__DLIB_FILE_DESCRIPTOR (configuration symbol) . . . 85
__eeprom (extended keyword) 232, 242
__embedded_cplusplus (predefined symbol) 271
__enable_interrupt (intrinsic function) 266
__exit (library function) 79
__far (extended keyword) 232, 242
__far_func (extended keyword) 243
__far_func (function pointer) 231
__FILE__ (predefined symbol). 271
__FUNCTION__ (predefined symbol) 124, 271
__func__ (predefined symbol) 124, 271
__gets, in stdio.h. 281
__get_interrupt_state (intrinsic function) 266
__halt (intrinsic function) 267
__huge (extended keyword) 232, 243
__huge_func (extended keyword). 244
__huge_func (function pointer). 232
__iar_maximum_atexit_calls 58
__IAR_SYSTEMS_ICC__ (predefined symbol) 271
__ICCSTM8__ (predefined symbol). 271
__interrupt (extended keyword) 38, 244
  using in pragma directives 263
__intrinsic (extended keyword). 244
__LINE__ (predefined symbol) 271
__LITTLE_ENDIAN__ (predefined symbol). 271
__low_level_init 77
  initialization phase 14
__low_level_init, customizing 80
__lseek (library function) 86
__memory_of, operator. 131
__monitor (extended keyword) 245
__near (extended keyword). 232, 245
__near_func (extended keyword) 245

__near_func (function pointer) 231
__noreturn (extended keyword) 246
__no_init (extended keyword) 163, 246
__no_operation (intrinsic function). 267
__open (library function) 86
__PRETTY_FUNCTION__ (predefined symbol). 271
__printf_args (pragma directive). 259
__program_start (label). 77
__read (library function). 86
customizing 82
__ReportAssert (library function) 91
__root (extended keyword) 246
__rt_version (runtime model attribute) 93
__scanf_args (pragma directive) 261
__section_begin (extended operator) 121
__section_end (extended operator) 121
__section_size (extended operator). 121
__set_interrupt_state (intrinsic function) 267
__STDC_VERSION__ (predefined symbol) 272
__STDC__ (predefined symbol) 272
__SUBVERSION__ (predefined symbol). 272
__task (extended keyword) 247
__TIME__ (predefined symbol) 272
__tiny (extended keyword) 232, 247
__trap (intrinsic function) 267
__ungetchar, in stdio.h 281
__VA_ARGS__ (preprocessor extension). 117
__VER__ (predefined symbol) 272
__wait_for_interrupt (intrinsic function). 267
__weak (extended keyword) 248
__write (library function) 86
customizing 82
__write_array, in stdio.h 281
__write_buffered (DLIB library function). 73
-D (compiler option) 184
-d (iarchive option) 332
-e (compiler option) 190
-f (compiler option). 191
-f (iobjmanip option). 333
-f (linker option) 214
-I (compiler option). 192
-l (compiler option). 192
for creating skeleton code 99
-O (compiler option) 198
-o (iarchive option) 335
-o (ielfdump option) 335
-o (linker option). 220
-r (iarchive option) 338
-S (iarchive option) 340
-s (ielfdump option) 339
-t (iarchive option) 342
-V (iarchive option). 342
-x (iarchive option) 333
--all (ielfdump option) 329
--bin (ielftool option) 329
--char_is_signed (compiler option) 183
--char_is_unsigned (compiler option) 183
--checksum (ielftool option) 330
--code (ielfdump option) 331
--code_model (compiler option) 183
--config (linker option) 209
--config_def (linker option). 209
--core (compiler option) 184
--cpp_init_routine (linker option) 210
--create (iarchive option). 331
--c89 (compiler option). 182
--data_model (compiler option) 185
--debug (compiler option). 185
--debug_lib (linker option) 210
--define_symbol (linker option) 210
--delete (iarchive option). 332
--dependencies (compiler option) 185
--dependencies (linker option) 211
--diagnostics_tables (compiler option) 188
--diagnostics_tables (linker option). 213
--diag_error (compiler option) 186
--diag_error (linker option) 211
--diag_remark (compiler option). 187

--diag_remark (linker option) 212
--diag_suppress (compiler option) 187
--diag_suppress (linker option) 212
--diag_warning (compiler option) 188
--diag_warning (linker option) 213
--discard_unused_publics (compiler option) 188
--dlib_config (compiler option) 189
--ec++ (compiler option) 190
--edit (isymexport option) 332
--eec++ (compiler option) 190
--enable_multibytes (compiler option) 190
--entry (linker option) 213
--error_limit (compiler option) 191
--error_limit (linker option) 214
--export_builtin_config (linker option) 214
--extract (iarchive option) 333
--fill (ielftool option) 334
--force_output (linker option) 215
--header_context (compiler option) 192
--ihex (ielftool option) 334
--image_input (linker option) 215
--keep (linker option) 216
--log (linker option) 216
--log_file (linker option) 216
--mangled_names_in_messages (linker option) 217
--map (linker option) 217
--mfc (compiler option) 193
--misrac_verbose (compiler option) 181
--misrac_verbose (linker option) 208
--misrac1998 (compiler option) 181
--misrac1998 (linker option) 208
--misrac2004 (compiler option) 181
--misrac2004 (linker option) 208
--no_code_motion (compiler option) 194
--no_cross_call (compiler option) 194
--no_cse (compiler option) 194
--no_inline (compiler option) 195
--no_library_search (linker option) 218
--no_locals (linker option) 219
--no_path_in_file_macros (compiler option) 195
--no_range_reservations (linker option) 219
--no_remove (linker option) 219
--no_static_destruction (compiler option) 196
--no_system_include (compiler option) 196
--no_tbaa (compiler option) 196
--no_typedefs_in_diagnostics (compiler option) 197
--no_unroll (compiler option) 197
--no_warnings (compiler option) 198
--no_warnings (linker option) 219
--no_wrap_diagnostics (compiler option) 198
--no_wrap_diagnostics (linker option) 220
--only_stdout (compiler option) 199
--only_stdout (linker option) 220
--output (compiler option) 199
--output (iarchive option) 335
--output (ielfdump option) 335
--output (linker option) 220
--place_holder (linker option) 220
--predef_macro (compiler option) 200
--preinclude (compiler option) 200
--preprocess (compiler option) 200
--ram_reserve_ranges (isymexport option) 335
--raw (ielfdump] option) 336
--redirect (linker option) 221
--relaxed_fp (compiler option) 201
--remarks (compiler option) 202
--remarks (linker option) 221
--remove_section (iobjmanip option) 336
--rename_section (iobjmanip option) 337
--rename_symbol (iobjmanip option) 337
--replace (iarchive option) 338
--require_prototypes (compiler option) 202
--reserve_ranges (isymexport option) 338
--section (ielfdump option) 339
--silent (compiler option) 202
--silent (iarchive option) 340
--silent (ielftool option) 340
--silent (linker option) 222

--simple (ielftool option) 340
--srec (ielftool option) 340
--srec-len (ielftool option) 341
--srec-s3only (ielftool option) 341
--strict (compiler option) 203
--strip (ielftool option) 341
--strip (iobjmanip option) 341
--strip (linker option) 222
--strtab (ielfdump option) 335, 339
--symbols (iarchive option) 342
--system_include_dir (compiler option) 203
--toc (iarchive option) 342
--use_unix_directory_separators (compiler option) 203
--verbose (iarchive option) 342
--verbose (ielftool option) 342
--vla (compiler option) 204
--warnings_affect_exit_code (compiler option) 171, 204
--warnings_affect_exit_code (linker option) 222
--warnings_are_errors (compiler option) 205
--warnings_are_errors (linker option) 222
.comment (ELF section) 306
.debug (ELF section) 306
.eeprom.noinit (section) 307
.far_func.text (section) 309
.far.bss (section) 307
.far.data (section) 308
.far.data_init (section) 308
.far.noinit (section) 308
.far.rodata (section) 308
.huge_func.text (section) 310
.huge.bss (section) 309
.huge.data (section) 309
.huge.data_init (section) 310
.huge.noinit (section) 310
.huge.rodata (section) 310
.iar.debug (ELF section) 306
.iar.dynexit (section) 310
.intvec (section) 311
.near_func.text (section) 312
.near.bss (section) 311
.near.data (section) 311
.near.data_init (section) 311
.near.noinit (section) 311
.near.rodata (section) 312
.rel (ELF section) 306
.rela (ELF section) 306
.shstrtab (ELF section) 306
.strtab (ELF section) 306
.symtab (ELF section) 306
.tiny.bss (section) 312
.tiny.data (section) 312
.tiny.data_init (section) 313
.tiny.noinit (section) 313
.tiny.rodata (section) 313
.vregs (section) 314
@ (operator)
placing at absolute address 151
placing in sections 152
#include files, specifying 169, 192
#warning message (preprocessor extension) 273
%Z replacement string,
implementation-defined behavior 358

Numerics

16-bit pointers, accessing memory 25
24-bit pointers, accessing memory 25
32-bit data types, avoiding 147
32-bits (floating-point format) 230