IAR C Compiler

Reference Guide

for Freescale's

S08 Microprocessor Family

COPYRIGHT NOTICE

Copyright © 2008 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.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc.

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

EDITION NOTICE

First edition: June 2008

Part number: CS08-1

This guide applies to version 1.x of IAR Embedded Workbench® for S08.

Internal reference: T7.6; 5.2, IJOA

Brief contents

Tables xvii

Preface xix

Part 1. Using the compiler 1

Getting started 3

Data storage 11

Functions 19

Placing code and data 25

The DLIB runtime environment 37

Assembler language interface 65

Efficient coding for embedded applications 77

Part 2. Reference information 93

External interface details 95

Compiler options 101

Data representation 125

Compiler extensions 133

Extended keywords 143

Pragma directives 153

Intrinsic functions 167

The preprocessor 171

Library functions 177

Segment reference 183

Implementation-defined behavior 191
Index 203

Contents

Tables xvii

Preface xix

Who should read this guide xix
How to use this guide xix
What this guide contains xx
Other documentation xxi
Further reading xxi
Document conventions xxii
Typographic conventions xxii
Naming conventions xxiii

Part 1. Using the compiler 1

Getting started 3

IAR language overview 3
Supported S08 devices 3
Building applications—an overview 4
Compiling 4
Linking 4
Basic settings for project configuration 5
Processor configuration 5
Code model 6
Optimization for speed and size 6
Runtime environment 6
Special support for embedded systems 8
Extended keywords 8
Pragma directives 8
Predefined symbols 8
Special function types 9
Accessing low-level features 9

Data storage 11
Introduction 11
Different ways to store data 11
Memory types 12
Data8 12
Data16 13
Using data memory attributes 13
Pointers and memory types 14
Structures and memory types 15
More examples 15
Auto variables on the stack 16
The stack 16
Dynamic memory on the heap 17
Functions 19
Function-related extensions 19
Code models and function memory attributes 19
Using code models 19
Using function memory attributes 20
Code banking 20
Performance 21
Code that cannot be banked 21
Mixing banked and non-banked code 21
Size limits for banked functions 21
Primitives for interrupts, concurrency, and OS-related programming 22
Interrupt functions 22
Monitor functions 23
Placing code and data 25
Segments and memory 25
What is a segment? 25
Placing segments in memory 26
Customizing the linker command file 26

Data segments 28
Static memory segments 29
The stack 32
The heap 33
Located data 34
Code segments 34
Startup code 34
Normal code 35
Interrupt vectors 35
Verifying the linked result of code and data placement 35
Segment too long errors and range errors 35
Linker map file 36
The DLIB runtime environment 37
Introduction to the runtime environment 37
Runtime environment functionality 37
Library selection 38
Situations that require library building 38
Library configurations 39
Debug support in the runtime library 39
Using a prebuilt library 40
Customizing a prebuilt library without rebuilding 42
Choosing formatters for printf and scanf 42
Choosing printf formatter 42
Choosing scanf formatter 43
Overriding library modules 44
Building and using a customized library 46
Setting up a library project 46
Modifying the library functionality 46
Using a customized library 47
System startup and termination 47
System startup 47
System termination 49

Customizing system initialization 50
Modifying the file cstartup.s78 50
Standard streams for input and output 51
Implementing low-level character input and output 51
Configuration symbols for printf and scanf 52
Customizing formatting capabilities 53
File input and output 54
Locale 54
Locale support in prebuilt libraries 55
Customizing the locale support 55
Changing locales at runtime 56
Environment interaction 57
Signal and raise 57
Time 58
Strtod 58
Assert 58
C-SPY runtime interface 59
Low-level debugger runtime interface 59
The debugger terminal I/O window 60
Checking module consistency 61
Runtime model attributes 61
Using runtime model attributes 62
Predefined runtime attributes 62
User-defined runtime model attributes 63
Assembler language interface 65
Mixing C and assembler 65
Intrinsic functions 65
Mixing C and assembler modules 66
Inline assembler 67
Calling assembler routines from C 68
Creating skeleton code 68
Compiling the code 69

Calling convention 70
Function declarations 70
Virtual registers 70
Preserved versus scratch registers 71
Function entrance 71
Function exit 72
Banked function calls 72
Restrictions for special function types 73
Examples 73
Function directives 74
Call frame information 75
Efficient coding for embedded applications 77
Selecting data types 77
Using efficient data types 77
Using the best pointer type 78
Anonymous structs and unions 78
Controlling data and function placement in memory 79
Data placement at an absolute location 80
Data and function placement in segments 81
Controlling compiler optimizations 83
Scope for performed optimizations 83
Optimization levels 84
Speed versus size 84
Fine-tuning enabled transformations 85
Writing efficient code 87
Saving stack space and RAM memory 88
Function prototypes 88
Integer types and bit negation 89
Protecting simultaneously accessed variables 89
Accessing special function registers 90
Non-initialized variables 91

Part 2. Reference information 93

External interface details 95

Invocation syntax 95

Compiler invocation syntax 95

Passing options 95

Environment variables 96

Include file search procedure 96

Compiler output 97

Diagnostics 98

Message format 98

Severity levels 99

Setting the severity level 100

Internal error 100

Compiler options 101

Options syntax 101

Types of options 101

Rules for specifying parameters 101

Summary of compiler options 103

Descriptions of options 105

--char_is_signed 106

--code_model 106

--core 106

-D 107

--debug, -r 107

--dependencies 108

--diag_error 109

--diag_remark 109

--diag_suppress 110

--diag_warning 110

--diagnostics_tables 110

--discard_unused_publics 111

--dlib_config 111

-e 112
--enable_multibytes 112
--error_limit 112
-f 113
--header_context 113
-I 113
-l 114
--library_module 115
--mfc 115
--module_name 115
--no_code_motion 116
--no_cross_call 116
--no_cse 116
--no_inline 117
--no_path_in_file_macros 117
--no_tbaa 117
--no_typedefs_in_diagnostics 118
--no_unroll 118
--no_warnings 119
--no_wrap_diagnostics 119
-O 119
-o, --output 120
--omit_types 120
--only_stdout 121
--output, -o 121
--preinclude 121
--preprocess 122
--public_equ 122
-r, --debug 122
--remarks 123
--require_prototypes 123
--silent 123
--strict_ansi 124
--warnings_affect_exit_code 124

--warnings_are_errors 124

Data representation 125

Alignment 125

Alignment on the S08 microcontroller 125

Basic data types 126

Integer types 126

Floating-point types 127

Pointer types 128

Function pointers 128

Data pointers 129

Casting 129

Structure types 130

General layout 130

Type qualifiers 131

Declaring objects volatile 131

Declaring objects const 132

Compiler extensions 133

Compiler extensions overview 133

Enabling language extensions 134

C language extensions 134

Important language extensions 134

Useful language extensions 136

Minor language extensions 139

Extended keywords 143

General syntax rules for extended keywords 143

Type attributes 143

Object attributes 146

Summary of extended keywords 147

Descriptions of extended keywords 147

__banked 147

__data8 148

__data16 148

__interrupt ... 149
__intrinsic ... 149
__monitor ... 149
__no_init ... 150
__non_banked ... 150
__noreturn ... 150
__root ... 151
__task ... 151

Pragma directives ... 153

Summary of pragma directives ... 153
Descriptions of pragma directives ... 154
bitfields ... 154
constseg ... 155
data_alignment ... 155
dataseg ... 156
diag_default ... 156
diag_error ... 157
diag_remark ... 157
diag_suppress ... 157
diag_warning ... 158
include_alias ... 158
inline ... 159
language ... 159
location ... 160
message ... 160
object_attribute ... 161
optimize ... 161
__printf_args ... 162
required ... 162
rtmodel ... 163
__scanf_args ... 164
segment ... 164
type_attribute ... 165

vector ... 165

Intrinsic functions ... 167

Summary of intrinsic functions ... 167

Descriptions of intrinsic functions ... 168

__BGND ... 168

__disable_interrupt ... 168

__enable_interrupt ... 168

__get_interrupt_state ... 168

__illegal_opcode ... 169

__no_operation ... 169

__set_interrupt_state ... 169

__software_interrupt ... 169

__stop ... 169

__wait_for_interrupt ... 169

The preprocessor ... 171

Overview of the preprocessor ... 171

Descriptions of predefined preprocessor symbols ... 172

Descriptions of miscellaneous preprocessor extensions ... 173

NDEBUG ... 174

_Pragma() ... 174

#warning message ... 174

__VA_ARGS__ ... 175

Library functions ... 177

Introduction ... 177

Header files ... 177

Library object files ... 177

Reentrancy ... 178

IAR DLIB Library ... 178

C header files ... 179

Library functions as intrinsic functions ... 180

Added C functionality ... 180

Segment reference 183

Summary of segments 183

Descriptions of segments 184

BANKED 184

CHECKSUM 184

CODE 185

CSTACK 185

DATA8_AC 185

DATA8_AN 185

DATA8_I 186

DATA8_ID 186

DATA8_N 186

DATA8_Z 186

DATA16_AC 187

DATA16_AN 187

DATA16_C 187

DATA16_I 187

DATA16_ID 188

DATA16_N 188

DATA16_Z 188

HEAP 188

INTVEC 189

RCODE 189

Implementation-defined behavior 191

Descriptions of implementation-defined behavior 191

Translation 191

Environment 192

Identifiers 192

Characters 192

Integers 194

Floating point 194

Arrays and pointers 195

Registers 195

Structures, unions, enumerations, and bitfields 195
Qualifiers 196
Declarators 196
Statements 196
Preprocessing directives 196
IAR DLIB Library functions 198

Index 203

Tables

1: Typographic conventions used in this guide xxii
2: Naming conventions used in this guide xxiii
3: Command line options for specifying library and dependency files 7
4: Memory types and their corresponding memory attributes 13
5: Code models 19
6: Function memory attributes 20
7: XLINK segment memory types 26
8: Memory layout of a target system (example) 27
9: Memory types with corresponding segment groups 29
10: Segment name suffixes 30
11: Library configurations 39
12: Levels of debugging support in runtime libraries 40
13: Prebuilt libraries 40
14: Customizable items 42
15: Formatters for printf 43
16: Formatters for scanf 44
17: Descriptions of printf configuration symbols 53
18: Descriptions of scanf configuration symbols 53
19: Low-level I/O files 54
20: Functions with special meanings when linked with debug info 59
21: Example of runtime model attributes 61
22: Predefined runtime model attributes 62
23: Registers used for returning values 72
24: Call frame information resources defined in a names block 75
25: Compiler optimization levels 84
26: Compiler environment variables 96
27: Error return codes 98
28: Compiler options summary 103
29: Integer types 126
30: Function pointers 128
31: Data pointers 129

32: Extended keywords summary 147
33: Pragma directives summary 153
34: Intrinsic functions summary 167
35: Predefined symbols 172
36: Traditional standard C header files—DLIB 179
37: Segment summary 183
38: Message returned by strerror()—IAR DLIB library 201

Preface

Welcome to the *IAR C Compiler Reference Guide for S08*. The purpose of this guide is to provide you with detailed reference information that can help you to use the compiler to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

Who should read this guide

You should read this guide if you plan to develop an application using the C language for the S08 microcontroller and need to get detailed reference information on how to use the compiler. In addition, you should have a working knowledge of the following:

- The architecture and instruction set of the S08 microcontroller. Refer to the documentation from Freescale for information about the S08 microcontroller
- The 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 compiler, you should read *Part 1. Using the compiler* in this guide.

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

If you are new to using the IAR Systems build tools, we recommend that you first study the *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 compiler

- *Getting started* gives the information you need to get started using the compiler for efficiently developing your application.
- *Data storage* describes how data can be stored in memory, with emphasis on the different data memory attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization and introduces the file `cstartup`, as well as 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.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Reference information

- *External interface details* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler's diagnostic system works.
- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Compiler extensions* gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.
- *Extended keywords* gives reference information about each of the S08-specific keywords that are extensions to the standard C language.
- *Pragma directives* gives reference information about the pragma directives.

- *Intrinsic functions* gives reference information about the functions that can be used for accessing S08-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 library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *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 S08 microcontroller is described in a series of guides. For information about:

- Using the IDE with the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the IAR Assembler for S08, refer to the *IAR Assembler Reference Guide for S08*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library functions, refer to the online help system
- Using the MISRA C rules, refer to the *IAR Embedded Workbench® MISRA C Reference Guide*.

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

FURTHER READING

The following books may be of interest to you when using the IAR Systems development tools:

- *HCS08 Family Reference Manual*. Freescale Semiconductor, Inc.
- 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*. [The later editions describe the ANSI C standard.]
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C.* R&D Books.
- Mann, Bernhard. *C für Mikrocontroller.* Franzis-Verlag. [Written in German.]

We recommend that you visit the following web sites:

- The Freescale web site, **www.freescale.com**, contains information and news about the S08 microcontrollers.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.

Document conventions

When referring to a directory in your product installation for example `s08\doc`, the full path to the location is assumed, that is for example `c:\Program Files\IAR Systems\Embedded Workbench 5.0\s08\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.
`{option}`	A mandatory part of a command.
`a|b|c`	Alternatives in a command.
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.

Table 1: Typographic conventions used in this guide

Style	Used for
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

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

NAMING CONVENTIONS

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

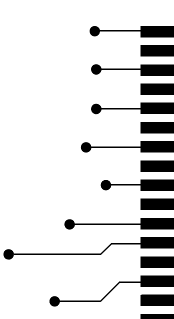
Brand name	Generic term
IAR Embedded Workbench® for S08	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for S08	the IDE
IAR C-SPY® Debugger for S08	C-SPY, the debugger
IAR C Compiler™ for S08	the compiler
IAR Assembler™ for S08	the assembler
IAR XLINK™ Linker	XLINK, the linker
IAR XAR Library builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide

Part 1. Using the compiler

This part of the IAR C Compiler Reference Guide for S08 includes the following chapters:

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



Getting started

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

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

Next, the compiler is introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the S08 microcontroller. In the following chapters, these techniques will be studied in more detail.

IAR language overview

The IAR C Compiler for S08 supports C, the most widely used high-level programming language in the embedded systems industry. Using the IAR C Compiler for S08, you can build freestanding applications that follow the standard ISO 9899:1990. This standard is commonly known as ANSI C.

The language 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 details, see the chapter *Compiler extensions*.

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *IAR Assembler Reference Guide for S08*.

Supported S08 devices

The IAR C Compiler for S08 supports all devices based on the Freescale S08 microcontroller. Devices based on the HC08 and RS08 architectures are *not* supported. The following extension is also supported:

- Banked code memory, using the `PPAGE` register. (Used for example in MC9S08QE128 devices.)

Note: In the chip documentation from Freescale Semiconductor, Inc., code *banks* are usually referred to as code *pages*.

Building applications—an overview

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

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

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

Below, the process for building an application on the command line is described. For information about how to build an application using the IDE, see the *IAR Embedded Workbench® IDE User Guide*.

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r78` using the default settings:

```
iccs08 myfile.c
```

In addition, you need to specify some critical options, see *Basic settings for project configuration*, page 5.

LINKING

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

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

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

```
xlink myfile.r78 myfile2.r78 -s __program_start -f
lnkmc9s08qe64.xcl dls08sf.r78 -o aout.a78 -r
```

In this example, `myfile.r78` and `myfile2.r78` are object files, `lnkmc9s08qe64.xcl` is the linker command file, and `dls08sf.r78` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

The linker produces output according to your specifications. Choose the output format that suits your purpose. You might want to load the output to a debugger—which means that you need output with debug information. Alternatively, you might want to load the output to a flash loader or a PROM programmer—in which case you need output without debug information, such as Intel-hex or Motorola S-records. The option `-F` can be used for specifying the output format. (The default output format is `motorola-s28`.)

Basic settings for project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler generate the best code for the S08 device you are using. You can specify the options either from the command line interface or in the IDE.

The basic settings are:

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

In addition to these settings, there are many other options and settings available for fine-tuning the result even further. For details about how to set options and for a list of all available options, see the chapters *Compiler 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 S08 microcontroller you are using.

Core (instruction set architecture)

The `--core` option is used for specifying the instruction set architecture of the S08 microcontroller you are using.

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

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

CODE MODEL

The compiler supports *code models* that you can use to control where in memory functions will be located. The following code models are available:

- The *Small* code model places functions in non-banked code memory by default
- The *Banked* code model places functions in banked code memory by default.

In the IDE, choose **Project>Options>General Options>Target>Code model**.

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

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

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 may also need to override certain library modules with your own customized versions.

The runtime library is the IAR DLIB Library, which supports ISO/ANSI 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 you choose can be one of the prebuilt libraries, or a library that you have customized and built yourself. The IDE provides a library project template that you can use for building your own library version. This gives you full control of the runtime environment. If your project only contains assembler source code, there is no need to choose a runtime library.

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

The way you set up a runtime environment and locate all the related files differs depending on which build interface you are using—the IDE or the command line.

Choosing a runtime library in the IDE

To choose a library, choose **Project>Options**, and click the **Library Configuration** tab in the **General Options** category. Choose the appropriate library from the **Library** drop-down menu.

Note that there are different configurations—Tiny, Normal, and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 39, 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.

Choosing runtime environment from the command line

Use the following command line options to specify the library and the dependency files:

Command line	Description
`-I s08\inc`	Specifies the include path to device-specific I/O definition files.
libraryfile`.r78`	Specifies the library object file
`--dlib_config C:\...\`*configfile*`.h`	Specifies the library configuration file

Table 3: Command line options for specifying library and dependency files

For a list of all prebuilt library object files, see Table 13, *Prebuilt libraries*, page 40. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 42.
- The size of the stack and the heap, see *The stack*, page 32, and *The heap*, page 33, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the S08 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 112 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 ISO/ANSI 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 model.

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

SPECIAL FUNCTION TYPES

The special hardware features of the S08 microcontroller are supported by the compiler's special interrupt functions. You can write a complete application without having to write any interrupt functions in assembler language.

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 22.

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 65.

Data storage

This chapter gives a brief introduction to the memory layout of the S08 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 memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concept of data memory types is described in relation to pointers, structures, and non-initialized memory. Finally, detailed information about data storage on the stack and the heap is provided.

Introduction

The IAR C Compiler for S08 can address 64 Kbytes of continuous data memory, ranging from `0x0000` to `0xFFFF`. 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 12.

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 and local variables declared `static`.

  In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Memory types*, page 12, and *Virtual registers*, page 70.

- Dynamically allocated data.

  An application can allocate data on the *heap*, where the data it 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 17.

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. By mapping 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 8-bit addressing and the memory accessed using 16-bit addressing are called data8 and data16 memory, respectively.

By default, your application uses the data16 memory type. However, it is possible to specify—for individual variables or pointers—the data8 memory type. 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.

DATA8

The data8 memory consists of the low 256 bytes of data memory. In hexadecimal notation, this is the address range `0x00-0xFF`.

A data8 object can only be placed in data8 memory, and the size of such an object is limited to 255 bytes. By using objects of this type, the code generated by the compiler to access them is minimized. This means a smaller footprint for the application, and faster execution at run-time.

DATA16

The data16 memory consists of the entire data memory. In hexadecimal notation, this is the address range `0x0000-0xFFFF`.

A data16 object can be placed anywhere in data16 memory, and the size of such an object is limited to 64 Kbytes–1.

USING DATA MEMORY ATTRIBUTES

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

The following table summarizes the available memory types and their corresponding keywords:

Memory type	Keyword	Address range	Pointer size
Data8	`__data8`	`0x0-0xFF`	1 byte
Data16 (default)	`__data16`	`0x0-0xFFFF`	2 bytes

Table 4: Memory types and their corresponding memory attributes

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

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 112 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 147.

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 143.

The following declarations place the variable `i` and `j` in data8 memory. The variables `k` and `l` will also be placed in data8 memory. The position of the keyword does not have any effect in this case:

```
__data8 int i, j;
int __data8 k, l;
```

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

In addition to the rules presented here—to place the keyword directly in the code—the directive `#pragma type_attribute` can 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. The following two declarations are equivalent:

```
typedef char __data8 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__data8 char b;
char __data8 *bp;
```

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 data16 memory is declared by:

```
int __data16 * p;
```

Note that the location of the pointer variable `p` is not affected by the keyword. In the following example, however, the pointer variable `p2` is placed in data8 memory. Like `p`, `p2` points to a character in data16 memory.

```
char __data16 * __data8 p2;
```

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.

For the IAR C Compiler for S08, the size of the `data8` and `data16` pointers are 8 and 16 bits, respectively.

Note: The natural data pointer size in the S08 instruction set is 16 bits. This means that accessing memory indirectly using `__data8` pointers is less efficient than using `__data16` pointers.

In the IAR C Compiler for S08, it is illegal, with one exception, to convert pointers between different types without explicit casts. See *Casting*, page 129.

STRUCTURES AND MEMORY TYPES

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

In the example below, the variable `gamma` is a structure placed in data16 memory.

```
struct MyStruct
{
  int alpha;
  int beta;
};
__data16 struct MyStruct gamma;
```

The following declaration is incorrect:

```
struct MySecondStruct
{
  int blue;
  __data16 int green;   /* Error! */
};
```

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 data8 memory is declared. The function returns a pointer to an integer in data16 memory. It makes no difference whether the memory attribute is placed before or after the data type. To read the following examples, start from the left and add one qualifier at each step

`int a;`	A variable defined in default memory.
`int __data8 b;`	A variable in data8 memory.
`__data16 int c;`	A variable in data16 memory.
`int * d;`	A pointer stored in default memory. The pointer points to an integer in default memory.

`int __data8 * e;`	A pointer stored in default memory. The pointer points to an integer in data8 memory.
`int __data8 * __data16 f;`	A pointer stored in data16 memory pointing to an integer stored in data8 memory.
`int __data16 * myFunction( int __data8 *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in data8 memory. The function returns a pointer to an integer stored in data16 memory.

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 small number of these variables might be 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 so-called *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 has returned. 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 ...
  return &x;
}
```

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`.

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 has not been 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 there is no piece of free memory that 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 the ISO/ANSI C standard, the compiler provides several extensions for writing functions in C. Using these, you can:

- Control the storage of functions in memory, using code models and memory attributes
- Use primitives for interrupts, concurrency, and OS-related programming
- Facilitate function optimization
- Access hardware features.

The compiler supports this by means of compiler options, extended keywords, pragma directives, and intrinsic functions.

For more information about optimizations, see *Writing efficient code*, page 87. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models and function memory attributes

Some S08 devices support *code banking* in order to extend the code memory beyond the limits imposed by the 64-Kbytes address space. Each function generated by the compiler is either *banked* or *non-banked*.

USING CODE MODELS

By means of code models, you can specify whether functions are banked or non-banked by default. These code models are available:

Code model name	Description
Small (default)	Functions are by default non-banked. This is the only available code model if you are using a device that does not support code banking.
Banked	Functions are by default banked.

Table 5: Code models

All user modules and library modules in the project must use the same code model.

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 106.

USING FUNCTION MEMORY ATTRIBUTES

For individual functions, you can override the default function type specified by the code model by using the appropriate *function memory attribute*. The following attributes are available:

Function memory attribute	Address range	Pointer size	Default in code model	Description
`__non_banked`	`0-0xFFFF`	2 bytes	Small	Non-banked function.
`__banked`	`0-0xFFFFFF`	3 bytes	Banked	Banked function.

Table 6: Function memory attributes

Banked and non-banked function pointers are incompatible: a banked function cannot be called with a non-banked function pointer, or vice versa. For more information about calling banked functions, see *Banked function calls*, page 72.

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

Code banking

Using code banking in the S08 compiler is simple, all you have to do is to specify the correct instruction set (`--core=V4`) and the Banked code model (`--code_model=banked`). This makes all functions banked by default. Any ISO/ANSI C-compliant program will continue to work as expected.

In the IDE, choose **Project>Options>General Options>Target** and set the **Device** and **Code model** options. This will automatically determine the instruction set architecture, the default linker command file, and the C-SPY® device description file.

You can also override the defaults given by the code model by specifying function memory attributes for individual functions. This way, you can make individual functions banked even if you are using the Small code model, or make individual functions non-banked if you are using the Banked code model.

PERFORMANCE

Because code banking is supported by the S08 hardware, the performance overhead is usually quite small. Non-banked calls are two cycles faster and require one less byte of code per call. The return from a banked function is the same size as a non-banked return, but takes one more cycle to execute.

However, *indirect banked calls* are not supported by the hardware, and are implemented by the compiler using a runtime library function. This takes three times as long as a non-banked indirect call, and requires 4 bytes of code rather than 1.

Pointers to banked functions are 3 bytes, but pointers to non-banked are only 2 bytes. To save data space, you might want to consider making functions that can be called via function pointers non-banked.

For these reasons, functions that are called inside a tight loop should be non-banked, to improve execution speed. Functions that are called from many different locations should also be non-banked, to improve code size. Finally, making all functions that are accessible through function pointers non-banked improves execution speed, code size, and data memory usage.

CODE THAT CANNOT BE BANKED

Some code cannot be banked, including interrupt service routines and selected parts of the runtime library. The compiler will automatically place such code in a segment that must be linked in non-banked code memory.

The following selected parts must be located in non-banked memory:

- Interrupt service routines (the segment `CODE`)
- The system startup code (the segment `RCODE`)
- Functions declared `__non_banked` (the segment `CODE`)
- Constants (the segment `DATA16_C`)
- Selected parts of the runtime library (the segment `CODE`).

MIXING BANKED AND NON-BANKED CODE

If you mix banked and non-banked functions, there are a few things to pay attention to.

It is not possible to call both banked and non-banked functions via the same function pointer: the function pointer type includes a function memory attribute which must match the attribute of the function it points to.

SIZE LIMITS FOR BANKED FUNCTIONS

Banked functions cannot be distributed over multiple flash memory pages. Therefore, the maximum size of a banked function is 16 Kbytes. Non-banked functions have no such restriction.

Certain aggressive size optimizations like cross call require that the functions to be optimized are located in the same flash page. This can cause large clusters of functions to be formed, which might not fit in single flash page. If this occurs, you can either disable the cross call optimization or repartition your code.

Primitives for interrupts, concurrency, and OS-related programming

The IAR C Compiler for S08 provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

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

INTERRUPT FUNCTIONS

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

In general, when an interrupt occurs in the code, the microcontroller simply stops executing the code it runs, and starts executing an interrupt routine instead. It is extremely important that the environment of the interrupted function is restored after the interrupt has been 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 when the code that handled the interrupt has been executed.

The S08 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 S08 microcontroller documentation from the chip manufacturer. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors.

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

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

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

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

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. You can also define an interrupt function without a vector, in which case you must populate the vector table by other means. See the chip manufacturer's S08 microcontroller documentation for more information about the interrupt vector table.

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 149.

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 semaphore is implemented using one static variable and two monitor functions. 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.

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

/* get_lock -- Try to lock the lock.
 * Return 1 on success and 0 on failure. */

__monitor int get_lock(void)
{
  if (the_lock == 0)
  {
    /* Success, we managed to lock the lock. */
    the_lock = 1;
    return 1;
  }
  else
  {
    /* Failure, someone else has locked the lock. */
```

```
    return 0;
  }
}


/* release_lock -- Unlock the lock. */

__monitor void release_lock(void)
{
  the_lock = 0;
}
```

The following is an example of a program fragment that uses the semaphore:

```
void my_program(void)
{
  if (get_lock())
  {
    /* ... Do something ... */

    /* When done, release the lock. */
    release_lock();
  }
}
```

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

A *segment* is a logical entity containing a piece of data or code that should be mapped to a physical location in memory. Each segment consists of many *segment parts*. Normally, each function or variable with static storage duration is placed in a segment part. A segment part is the smallest linkable unit, which allows the linker to include only those units that are referred to. The segment could be placed either in RAM or in ROM. Segments that are placed in RAM do not have any content, they only occupy space.

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

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

At compile time, the compiler assigns each segment its contents. The linker is responsible for placing the segments in the physical memory range, in accordance with the rules specified in the linker command file. There are ready-made linker command files, but, if necessary, they can be easily modified according to the requirements of your target system and application. It is important to remember that, from the linker's point of view, all segments are equal; they are simply named parts of memory.

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

Segment memory type

XLINK assigns a segment memory type to each of the segments. In some cases, the individual segments have the same name as the segment memory type they belong to, for example `CODE`. Make sure not to confuse the individual segment names with the segment memory types in those cases.

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

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

Table 7: XLINK segment memory types

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

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

Placing segments in memory

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

In particular, the linker command file specifies:

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

This section describes the methods for placing the segments in memory, which means that you can to customize the linker command file to suit the memory layout of your target system. For showing the methods, fictitious examples are used.

CUSTOMIZING THE LINKER COMMAND FILE

The `config` directory contains ready-made linker command files for all supported devices (filename extension `xcl`). The files contain the information required by the linker, and is ready to be used.

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

Range	Type
0x0080–0x17FF	RAM
0x1880–0x207F	RAM
0x2080–0x7FFF	ROM
0xC000–0xFFFF	ROM

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

The ROM can be used for storing `CONST` and `CODE` segment memory types. The RAM memory can contain segments of `DATA` type.

Remember not to change the original file. We recommend that you make a copy in the working directory, and modify the copy instead.

The contents of the linker command file

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

- The CPU used:

  `-cs08`

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

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

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

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

Using the -Z command for sequential placement

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

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x2080-0x7FFF`.

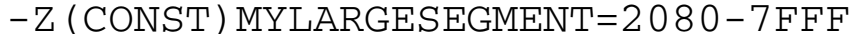
```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=2080-7FFF
```

Two segments of different types can be placed consecutively in the same memory area by not specifying a range for the second segment. In the following example, the `MYSEGMENTA` segment is first located in memory. Then, the rest of the memory range could be used by `MYCODE`.

```
-Z(CONST)MYSEGMENTA=2080-7FFF
-Z(CODE)MYCODE
```

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

```
-Z(CONST)MYSMALLSEGMENT=2080-217F
-Z(CONST)MYLARGESEGMENT=2080-7FFF
```



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

Using the -P command for packed placement

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

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

```
-P(DATA)MYDATA=0080-17FF
```

If your application has an additional RAM area in the memory range `0x1880-0x207F`, you can simply add that to the original definition:

```
-P(DATA)MYDATA=0080-17FF,1880–207F
```

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

Data segments

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

To get a clear understanding about how the data segments work, you must be familiar with the different memory types available in the compiler. If you need to refresh these details, see the chapter *Data storage*.

STATIC MEMORY SEGMENTS

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

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

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

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

Segment naming

The names of the segments consist of two parts—the segment group name and a *suffix*—for instance, `DATA16_Z`. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for example `DATA16` and `__data16`. The following table summarizes the memory types and the corresponding segment groups:

Memory type	Segment group	Memory range
Data8	`DATA8`	`0x00-0xFF`
Data16	`DATA16`	`0x0000-0xFFFF`

Table 9: Memory types with corresponding segment groups

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the

XLINK segment memory type of each segment. For more details about segment memory types, see *Segment memory type*, page 26.

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

Categories of declared data	Suffix	Segment memory type
Non-initialized data	`N`	`DATA`
Zero-initialized data	`Z`	`DATA`
Non-zero initialized data	`I`	`DATA`
Initializers for the above	`ID`	`CONST`
Constants	`C`	`CONST`
Non-initialized absolute addressed data	`AN`	
Constant absolute addressed data	`AC`	

Table 10: Segment name suffixes

For a list of all supported segments, see *Summary of segments*, page 183.

Note: Because there is no ROM in the range `0x00–0xFF`, constants in data8 memory are implemented as initialized variables placed in the `DATA8_I` and `DATA8_ID` segments.

Examples

The following examples demonstrates how declared data is assigned to specific segments:

```
__data8 int j;
__data8 int i = 0;
```

The data8 variables that are to be initialized to zero when the system starts will be placed in the segment `DATA8_Z`.

```
__no_init __data8 int j;
```

The data8 non-initialized variables will be placed in the segment `DATA8_N`.

```
__data8 int j = 4;
```

The data8 non-zero initialized variables will be placed in the segment `DATA8_I` in RAM, and the corresponding initializer data in the segment `DATA8_ID` in ROM.

Initialized data

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

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

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

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

For example, if the segments are assigned the following ranges, the copy will fail:

`DATA16_I`	`0x1000-0x10FF` and `0x1200-0x12FF`
`DATA16_ID`	`0x4000-0x41FF`

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

`DATA16_I`	`0x1000-0x10FF` and `0x1200-0x12FF`
`DATA16_ID`	`0x4000-0x40FF` and `0x4200-0x42FF`

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

Data segments for static memory in the default linker command file

A typical linker command file contains the following directives to place the static data segments:

```
-P(DATA)DATA8_I,DATA8_Z,DATA8_N=80-FF
-P(DATA)DATA16_I,DATA16_Z,DATA16_N=80-17FF,1880-207F

-P(CONST)DATA8_ID,DATA16_ID,DATA16_C=2080-7FFF,C000-FFAD,FFC0-FFFF
```

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

THE STACK

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 segment used for holding the stack is called `CSTACK`. The system startup code initializes the stack pointer to the end of the stack segment.

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

Stack size allocation in the IDE

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

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

Stack size allocation from the command line

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

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

```
-D_CSTACK_SIZE=200
```

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

Specify an appropriate size for your application. Note that the size is written hexadecimally without the `0x` notation.

Placement of stack segment

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

```
-Z(DATA)CSTACK+_CSTACK_SIZE#207F
```

Note:

- This range does not specify the size of the stack; it specifies the range of the available memory
- The `#` allocates the `CSTACK` segment at the end of the memory area. In practice, this means that the stack will get all remaining memory at the same time as it is guaranteed that it will be at least `_CSTACK_SIZE` bytes in size.

Stack size considerations

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

THE HEAP

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

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

- The linker segment used for the heap
- Allocating the heap size, which differs depending on which build interface you are using
- Placing the heap segment in memory.

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

Heap size allocation in the IDE

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

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

Heap size allocation from the command line

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

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

```
-D_HEAP_SIZE=200
```

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

Specify the appropriate size for your application.

Placement of heap segment

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

```
-Z(DATA)HEAP+_HEAP_SIZE=80-17FF
```

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

Heap size and standard I/O

If you have 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 C-SPY simulator driver, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an S08 microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

LOCATED DATA

A variable that has been explicitly placed at an address, for example by using the `#pragma location` directive or the `@` syntax, will be placed in one of the data segments with an `_AC` or `_AN` suffix. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker command file.

If you create your own segments, these must also be defined in the linker command file using the `-Z` or `-P` segment control directives.

Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For a complete list of all segments, see *Summary of segments*, page 183.

STARTUP CODE

The segment `RCODE` contains code used during system setup and runtime initialization (`cstartup`), and during system termination (`cexit`). Because the contents of this segment must be placed into one continuous block of memory, the `-P` segment directive cannot be used. The `cstartup.s78` file automatically populates the reset vector with the program start address.

In a typical linker command file, the following line places the RCODE segment in memory:

```
-Z(CODE)RCODE=2080-7FFF,C000-FFAD,FFC0-FFFF
```

NORMAL CODE

Functions declared without a memory type attribute are placed in different segments, depending on which code model you are using.

If you use the Small code model, or if the function is explicitly declared `__non_banked`, the code is placed in the `CODE` segment. If you use the Banked code model, or if the function is explicitly declared `__banked`, the code is placed in the `BANKED` segment.

In a typical linker command file, the following lines place the `CODE` and `BANKED` segments in memory:

```
-P(CODE)CODE=2080-7FFF,C000-FFAD,FFC0-FFFF
-P(CODE)BANKED=[8000-BFFF]*8+10000
```

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

Note: The compiler assumes that the `CODE` segment is linked in non-banked memory. The `BANKED` segment, on the other hand, can be placed in either banked or non-banked memory.

INTERRUPT VECTORS

The interrupt vector table contains pointers to interrupt service routines, including the reset vector. These pointers are placed at absolute addresses by the compiler.

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

LINKER MAP FILE

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

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

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

Normally, XLINK will not generate an output file if there are any errors, such as range errors, during the linking process. Use the option **Range checks disabled** in the IDE, or the option `-R` on the command line, to generate an output file even if a range error was encountered.

For further information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IAR Embedded Workbench® IDE User Guide*.

The DLIB runtime environment

This chapter describes the runtime environment in which an application executes. In particular, the chapter covers the DLIB runtime library and how you can modify it—setting options, overriding default library modules, or building your own library—to optimize it for your application.

The chapter also covers system initialization and termination; how an application can control what happens before the function main is called, and how you can customize the initialization.

The chapter then describes how to configure functionality like locale and file I/O, how to get C-SPY® runtime support, and how to prevent incompatible modules from being linked together.

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. The DLIB runtime environment can be used as is together with the debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

The *runtime environment* supports ISO/ANSI C, including the standard template library. The runtime environment consists of the *runtime library*, which contains the functions defined by these standards, and include files that define the library interface.

The runtime library is delivered both as prebuilt libraries and as source files, and you can find them in the product subdirectories `s08\lib` and `s08\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 register 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.
- Special compiler support for some functions, for instance functions for floating-point arithmetics.

The runtime environment support as well as the size of the heap must be tailored for the specific hardware and application requirements.

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

LIBRARY SELECTION

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.

IAR Embedded Workbench comes with a set of prebuilt runtime libraries. To get the required runtime environment, you can customize it by:

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s78`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: normal or full.

In addition, you can also make your own library configuration, but that requires that you *rebuild* the library. This allows you to get full control of the runtime environment.

Note: Your application project must be able to locate the library, include files, and the library configuration file.

SITUATIONS THAT REQUIRE LIBRARY BUILDING

Building a customized library is complex. You should therefore carefully consider whether it is really necessary.

You must build your own library when you want to define your own library configuration with support for locale, file descriptors, multibyte characters, et cetera.

For information about how to build a customized library, see *Building and using a customized library*, page 46.

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, as well as tailoring the system header files used when compiling your application. The less functionality you need in the runtime environment, the smaller it is.

The following DLIB library configurations are available:

Library configuration	Description
Tiny DLIB	No locale interface, C locale, no file descriptor support, no multibyte and wide characters, and no hex floats in `strtod`. The function `rand` requires less memory than the standard function. However, the quality of the pseudo-random number sequence is not as good as when using the normal scheme.
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in `printf` and `scanf`, and no hex floats in `strtod`.
Full DLIB	Full locale interface, C locale, file descriptor support, multibyte characters in `printf` and `scanf`, and hex floats in `strtod`.

Table 11: Library configurations

In addition to these configurations, 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 46.

The prebuilt libraries are based on the default configurations, see Table 13, *Prebuilt libraries*, page 40. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

DEBUG SUPPORT IN THE RUNTIME LIBRARY

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

The following table describes the different levels of debugging support:

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

Table 12: Levels of debugging support in runtime libraries

If you build your application project with the XLINK options **With runtime control modules** or **With I/O emulation modules**, certain functions in the library will be replaced by functions that communicate with the debugger. For further information, see *C-SPY runtime interface*, page 59.

To set linker options for debug support in the IDE, choose **Project>Options** and select the **Linker** category. On the **Output** page, select the appropriate **Format** option.

Using a prebuilt library

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

- Code model—Small or Banked
- Library configuration—Tiny, Normal, or Full.

The following prebuilt runtime libraries are available:

Library name	Target	Code model	Library configuration
`dls08st.r78`	`S08`	Small	Tiny
`dls08sn.r78`	`S08`	Small	Normal
`dls08sf.r78`	`S08`	Small	Full

Table 13: Prebuilt libraries

Library name	Target	Code model	Library configuration
dls08bt.r78	S08	Banked	Tiny
dls08bn.r78	S08	Banked	Normal
dls08bf.r78	S08	Banked	Full

Table 13: Prebuilt libraries (Continued)

The names of the libraries are constructed in the following way:

`<type><target><code_model><lib_config>.r78`

where

- `<type>` is `dl` for the DLIB runtime environment
- `<target>` is `s08`
- `<code_model>` is one of `s` or `b` for Small (non-banked) and Banked, respectively
- `<lib_config>` is `t`, `n`, or `f`, for Tiny, Normal, and Full, respectively.

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

The IDE 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, you must specify the following items to get the required runtime library:

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

  `dls08sn.r78`

- Specify the include paths for the compiler and assembler:

  `-I s08\inc\`

- Specify the library configuration file for the compiler:

  `--dlib_config C:\...\s08\lib\dls08sn.h`

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

You can find the library object files and the library configuration files in the subdirectory `s08\lib\`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the compiler can be used as is. However, it is possible to customize parts of a library without rebuilding it. There are two different methods:

- Setting options for:
  - Formatters used by `printf` and `scanf`
  - The sizes of the heap and the stack
- Overriding library modules with your own customized versions.

The following items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 42
Startup and termination code	*System startup and termination*, page 47
Low-level input and output	*Standard streams for input and output*, page 51
File input and output	*File input and output*, page 54
Low-level environment functions	*Environment interaction*, page 57
Low-level signal functions	*Signal and raise*, page 57
Low-level time functions	*Time*, page 58
Size of heaps, stacks, and segments	*Placing code and data*, page 25

Table 14: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 44.

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, it is possible to optimize these functions even further, see *Configuration symbols for printf and scanf*, page 52.

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 library.

The following table summarizes the capabilities of the different formatters:

Formatting capabilities	_PrintfFull	_PrintfLarge	_PrintfSmall	_PrintfTiny
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`	Yes	No	No	No
Floating-point specifiers `e`, `E`, `f`, `F`, `g`, and `G`	Yes	Yes	No	No
Field width and precision, including `*`	Yes	Yes	Yes	No
Conversion specifier `n`	Yes	Yes	No	No
Format flag space, `+`, `-`, `#`, and `0`	Yes	Yes	Yes	No
Length modifiers `h`, `l`, `L`, `s`, `t`, and `Z`	Yes	Yes	Yes	No

Table 15: 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 52.

Specifying the print formatter in the IDE

To use any other formatter than the default (Full), 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 (`_PrintfFull`), add one of the following lines in the linker command file you are using:

```
-e_PrintfLarge=_Printf
-e_PrintfSmall=_Printf
-e_PrintfTiny=_Printf
```

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 library.

The following table summarizes the capabilities of the different formatters:

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

Table 16: 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 52.

Specifying scanf formatter in the IDE

To use any other formatter than the default (Full), 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 (`_ScanfFull`), add one of the following lines in the linker command file you are using:

```
-e_ScanfLarge=_Scanf
-e_ScanfSmall=_Scanf
```

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 initialization. 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 `s08\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), 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. 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:

   ```
   iccs08 library_module
   ```

   This creates a replacement object module file named `library_module.r78`.

   Note: The code model, include paths, and the library configuration file must be the same for `library_module` as for the rest of your code.
4. Add `library_module.r78` to the XLINK command line, either directly or by using an extended linker command file, for example:

   ```
   xlink library_module dls08sn.r78
   ```

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

   Run XLINK to rebuild your application.

   This will use your version of `library_module.r78`, instead of the one in the library. For information about the XLINK options, see the *IAR Linker and Library Tools Reference Guide*.

Building and using a customized library

In some situations, see *Situations that require library building*, page 38, it is necessary to rebuild the library. In those cases you need to:

- 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.

Information about the build process is described in the *IAR Embedded Workbench® IDE User Guide*.

Note: It is possible to build IAR Embedded Workbench projects from the command line by using the IAR Command Line Build Utility (`iarbuild.exe`). However, no make or batch files for building the library from the command line are provided.

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 has Full library configuration, see Table 11, *Library configurations*, page 39.

In the IDE, modify the generic options in the created library project to suit your application, see *Basic settings for project configuration*, page 5.

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. In addition, your library has its own library configuration file `dls08`*`libraryname`*`.h`, which sets up that specific library with full library configuration. For more information, see Table 14, *Customizable items*, page 42.

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

Modifying the library configuration file

In your library project, open the file `dls08`*`libraryname`*`.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 have built your library, you must make sure to use it in your application project.

In the IDE you must perform the following steps:

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

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s78`, `cexit.s78`, and `low_level_init.c`, located in the `s08\src\lib` directory.

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 environment.

For the hardware initialization, it looks like this:

Figure 1: Target hardware initialization phase

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

For the C initialization, it looks like this:

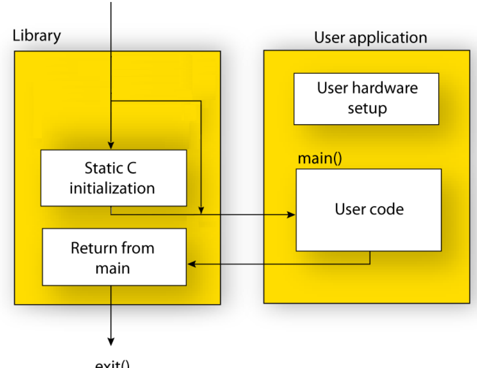


Figure 2: C initialization phase

- Static variables are initialized (if the return value of `__low_level_init` is non-zero). This clears zero-initialized memory or copies the values of other initialized variables from ROM to RAM memory. For more details, see *Initialized data*, page 31

- The `main` function is called, which starts the application.

SYSTEM TERMINATION

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

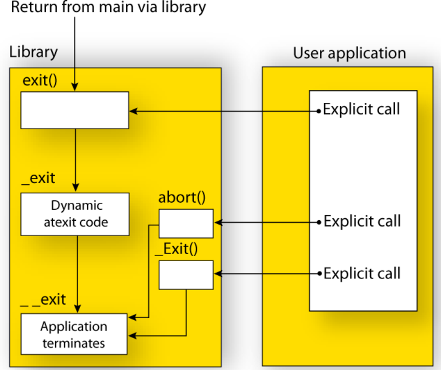


Figure 3: System termination phase

An application can terminate normally in two different ways:

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

As the ISO/ANSI 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 the following operations:

- Call functions registered to be executed when the application ends. This includes 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 perform anything extra at exit, for example resetting the system, you can write your own implementation of the `__exit(int)` function.

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY runtime interface*, page 59.

Customizing system initialization

The code for handling system startup is located in the source files `cstartup.s78` and `low_level_init.c`, located in the `s08\src\lib` directory.

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

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cstartup` before the data segments are initialized. Modifying the file `cstartup.s78` directly should be avoided.

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

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

Normally, there is no need for customizing the file `cexit.s78`.

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 46.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s78`, you do not have to rebuild the library.

MODIFYING THE FILE CSTARTUP.S78

As noted earlier, you should not modify the file `cstartup.s78` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s78`, 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 44.

Standard streams for input and output

There are three standard communication channels (streams)—`stdin`, `stdout`, and `stderr`—which are defined in `stdio.h`. If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you need to customize the low-level functionality to suit your hardware.

There are primitive I/O functions, which are the fundamental functions through which 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 `s08\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 46. 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 *C-SPY runtime interface*, page 59.

Example of using __write and __read

The code in the following examples use memory-mapped I/O to write to an LCD display:

```
__no_init volatile unsigned char LCD_IO @ address;

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)
  {
    LCD_IO = * Buf++;
    ++nChars;
  }
  return nChars;
}
```

Note: A call to `__write` with `BUF` having the value `NULL` is a command to flush the handle.

The code in the following example uses memory-mapped I/O to read from a keyboard:

```
__no_init volatile unsigned char KB_IO @ address;

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 = KB_IO;
    if (c == 0)
      break;
    *Buf++ = c;
    ++nChars;
  }
  return nChars;
}
```

For information about the `@` operator, see *Controlling data and function placement in memory, page 79*.

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 42.

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

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

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

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

Table 17: Descriptions of printf configuration symbols

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

Scanf configuration symbols	Includes support for
_DLIB_SCANF_MULTIBYTE	Multibyte characters
_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 18: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 46. 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 need to customize them to suit your hardware. In order 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 a number of characters.

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 39. 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 the following 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 19: 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 will be linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 39.

Locale

Locale is a part of the C language that allows language- and country-specific settings for a number of 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 during runtime.
- Libraries with *normal library configuration* do not have support for the locale interface.

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

CUSTOMIZING THE LOCALE SUPPORT

If you decide to rebuild the library, you can choose between the following 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    /* US english */
#define _LOCALE_USE_EN_GB    /* UK 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 46.

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 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
```

The last string must be empty. 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 `s08\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 44.

If you need to use the `system` function, you need to 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 46.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` will be replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 39.

Signal and raise

There are default implementations of the functions `signal` and `raise` 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 s08\src\lib directory. For information about overriding default library modules, see *Overriding library modules*, page 44.

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 46.

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 s08\src\lib directory. For information about overriding default library modules, see *Overriding library modules*, page 44.

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 46.

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 will be replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY runtime interface*, page 59.

Strtod

The function strtod does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you need to rebuild the library, see *Building and using a customized library*, page 46. Enable the configuration symbol _DLIB_STRTOD_HEX_FLOAT in the library configuration file.

Assert

If you have linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file xreportassert.c to your application project. Alternatively, you can rebuild the library. The __ReportAssert function generates the assert notification. You can find template code in the s08\src\lib directory. For further information, see *Building and using a customized library*, page 46. 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.

C-SPY runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 39.

In this case, C-SPY variants of the following library functions will be linked to the application:

Function	Description
abort	C-SPY 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 has been reached *
__open	Opens a file on the host computer
__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 20: Functions with special meanings when linked with debug info

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

LOW-LEVEL DEBUGGER RUNTIME INTERFACE

The low-level debugger runtime interface is used for communication between the application being debugged and the debugger itself. The debugger provides runtime services to the application via this 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 using file I/O before any flash file system I/O drivers have been 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 trace 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 have linked it with the XLINK 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.

THE DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Debug support in the runtime library*, page 39. 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` has been included in the DLIB library. This module buffers the output and sends it to the debugger one line at a time, speeding up the output. Note that this function uses about 80 bytes of RAM memory.

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

```
-e__write_buffered=__write
```

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the compiler, assembler, and linker to ensure module consistency.

When developing an application, it is important to ensure that incompatible modules are not used together. For example, a module compiled using the Small code model cannot be used together with modules compiled using the Banked code model.

The tools provided by IAR Systems use a set of predefined runtime model attributes. You can use these predefined attributes or define your own to perform any type of consistency check.

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

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

Table 21: Example of runtime model attributes

USING RUNTIME MODEL ATTRIBUTES

Runtime model attributes can be specified in your C source code to ensure module consistency with other object files by using the `#pragma rtmodel` directive. For example:

```
#pragma rtmodel="__rt_version", "1"
```

For detailed syntax information, see *rtmodel*, page 163.

Runtime model attributes can also be specified in your assembler source code by using the `RTMODEL` assembler directive. For example:

```
  RTMODEL "color", "red"
```

For detailed syntax information, see the *IAR Assembler Reference Guide for S08*.

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

At link time, XLINK 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 and assembler code.

Runtime model attribute	Value	Description
`__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.
`__code_model`	`small` or `banked`	Corresponds to the code model used in the project.

Table 22: Predefined runtime model attributes

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

If you are using assembler routines in the C code, refer to the chapter *Assembler directives* in the *IAR Assembler Reference Guide for S08.*

USER-DEFINED RUNTIME MODEL ATTRIBUTES

In cases where the predefined runtime model attributes are not sufficient, you can define your own attributes by using the `RTMODEL` assembler directive. 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 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`. You should declare this in each module that assumes that the UART is in a particular mode. This is how it could look like in one of the modules:

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

Assembler language interface

When you develop an application for an embedded system, there may be situations where you will find it necessary to write parts of the code in assembler, for example when using mechanisms in the S08 microcontroller that require precise timing and special instruction sequences.

This chapter describes the available methods for this, as well as 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.

Finally, the chapter covers how functions are called in the two code models 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 compiler provides several ways to mix C and assembler:

- 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 small number of 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 modules. There are several benefits with this 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 functions.

There will be some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all registers are destroyed by an inline assembler instruction.

When an application is written partly in assembler language and partly in C, you are faced with a number of 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 68. The following two are covered in the section *Calling convention*, page 70.

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 75.

The recommended method for mixing C and assembler modules is described in *Calling assembler routines from C*, page 68.

INLINE ASSEMBLER

It is possible to insert assembler code directly into a C function. The `asm` keyword inserts 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.

```
bool __data8 flag;

void foo(void)
{
  while (!flag)
  {
    asm("MOV PIND,flag");
  }
}
```

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

The inline assembler instruction will simply be inserted at the given location in the program flow. The consequences or side-effects the insertion may have on the surrounding code have not been taken into consideration. If, for example, registers or memory locations are altered, they may 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 code. This makes the inline assembler code fragile, and will possibly also become a maintenance problem if you upgrade the compiler in the future. In addition, there are 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
- Auto variables cannot be accessed.

Inline assembler is therefore often best avoided. If there is no suitable intrinsic function available, we recommend the use of 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 is to 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 the following 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 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 the following options to compile the skeleton code:

```
iccs08 skeleton -lA .
```

The `-lA` option creates an assembler language output file including C source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C module (`skeleton`), but with the filename extension `s78`. Also remember to specify the code model you are using, a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s78`.

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, select **Project>Options>C Compiler>List** and deselect the suboption **Include compiler runtime 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.

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. The following items are examined:

- Function declarations
- Preserved versus scratch registers
- Function entrance
- Function exit
- Return address handling
- Banked function calls.

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 a_function(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.

VIRTUAL REGISTERS

The compiler can use *virtual* registers; that is, static memory locations used for storing variables and temporary values. These virtual registers are called `?vr0–?vr7`, and are allocated in the direct page by the linker if they are used by the compiler.

For convenience, the 1-byte registers are combined into 2-byte pairs, `?vw0–?vw3`, so that, for example, `?vw0` occupies the same space as `?vr0` and `?vr1`.

PRESERVED VERSUS SCRATCH REGISTERS

The general S08 CPU registers are divided into two separate sets, which are described in this section.

Scratch registers

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

The registers `A`, `HX`, and `?vr0–?vr3` are scratch registers.

Preserved registers

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

The virtual registers `?vr4–?vr7` are preserved across function calls.

FUNCTION ENTRANCE

Parameters are passed on the stack. The first parameter is pushed last (closest to the stack pointer).

Stack layout

The stack layout at function entry is described in Figure 4, *Stack image after the function call*:

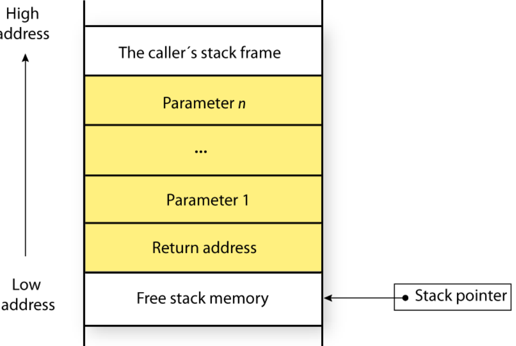


Figure 4: Stack image after the function call

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 scalar and floating-point values are `A` and `HX`.

Return values	Passed in registers
8-bit scalar values	`A`
16-bit scalar values	`HX`
24-bit function pointers	`A` (the page), `HX` (the address)

Table 23: Registers used for returning values

All other values—including `long` and `float` scalar values, `structs` and unions—are passed using a hidden return value pointer, an extra parameter passed to the function as the last parameter (pushed first). The called function writes the return value to the location pointed to by the return value pointer.

Stack layout at function exit

The calling function is responsible for cleaning up the parameters from the stack after the called function has returned.

Return address handling

A function written in assembler language should, when finished, return to the caller. The return address is automatically pushed on the stack after the parameters. The return address is 2 bytes in a `__non_banked` function and 3 bytes in a `__banked` function. This is the only calling convention difference between banked and non-banked functions.

BANKED FUNCTION CALLS

S08 devices that support code banking add two new instructions—`CALL` and `RTC`—to the instruction set to allow calling and returning from functions in banked memory. A 16-Kbyte paging window in the address space maps to different flash pages, depending on the current value of the `PPAGE` special function register. Banked functions are identified by 3-byte addresses in the form `page:addr`, where `addr` is an ordinary 2-byte address within the 16-Kbyte paging window.

Ordinary non-banked functions must be placed outside the paging window, and are always available regardless of the value of PPAGE. They are called with the JSR instruction, and return with RTS.

Unlike the non-banked JSR instruction, the CALL instruction does not accept an indirect pointer operand. Banked calls through function pointers can be implemented by an auxiliary routine like this (assuming the destination page is in A, and the address in HX):

```
call_ind:
        PSHX           ; push destination page:addr
        PSHH
        PSHA
        RTC            ; pop page:addr and jump
```

A CALL to this routine will push the return address and page, then transfer control to the destination function. Any banked function can be called like this.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

Interrupt handlers must preserve register H and all virtual registers (?vr0–?vr7). The registers A, X, SP, PC, and CCR are saved automatically by the hardware.

EXAMPLES

The following section shows a series of declaration examples and the corresponding calling conventions (assuming that the Small code model is being used). The complexity of the examples increases towards the end.

Example 1

Assume the following function declaration:

```
__non_banked char add1(char);
```

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

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

```
add1:
   lda     3,sp
   inca
   rts
```

Example 2

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

```
struct a_struct { int a; };
int a_function(struct a_struct x, int y);
```

The calling function pushes `y` and the `struct x` before calling `a_function`. The return value is passed in register `HX`.

Example 3

The function below will return a `struct`.

```
struct a_struct { int a; };
struct a_struct  a_function(int x);
```

The calling function must allocate a memory location for the return value, and pass a pointer to that memory as a hidden last parameter. The pointer is pushed first, then `x` is pushed. The called function writes the return value via the hidden pointer to the memory location allocated by the calling function.

Assume that the function instead would have been declared to return a pointer to the structure:

```
struct a_struct *  a_function(int x);
```

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed on the stack, and the return value is returned in register `HX`.

FUNCTION DIRECTIVES

Note: This type of directive is primarily intended to support static overlay, a feature which is useful in some microcontrollers. The IAR C Compiler for S08 does not use static overlay.

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

For reference information about the function directives, see the *IAR Assembler Reference Guide for S08*.

Call frame information

When debugging an application using C-SPY, it is possible to view the *call stack*, that is, the chain of functions that have called the current function. The compiler makes this possible by supplying 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 S08*.

The CFI directives will 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 may 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.

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

Resource	Description
`CFA`	The call frame base
`A`	Accumulator register
`H`	Index register, the most significant 8 bits
`X`	Index register, the least significant 8 bits
`?RET`	The return address
`?RETPAGE`	The return page
`SP`	The stack pointer
`?vr0–?vr7`	Virtual registers

Table 24: Call frame information resources defined in a names block

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
- Writing efficient code.

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.
- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array. For data8 the index type is `signed char` and for data16 it is `int`.
- Using floating-point types on a microprocessor without a math co-processor is very inefficient, both in terms of code size and execution speed.

- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

For details about representation of supported data types, pointers, and structures types, see the chapter *Data representation*.

USING THE BEST POINTER TYPE

Because `__data8` pointers are less efficient than `__data16` pointers, `__data8` pointers should only be used to save space in `struct`s, arrays, etc.

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 Compiler for S08, 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 112, for additional information.

Example

In the following example, the members in the anonymous `union` can be accessed, in function `f`, without explicitly specifying the `union` name:

```
struct s
{
  char tag;
  union
  {
    long l;
    float f;
  };
} st;

void f(void)
{
  st.l = 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 the following example:

```
__no_init volatile
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char way: 1;
    unsigned char out: 1;
  };
} @ address;
```

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

The following example illustrates how variables declared this way can be used:

```
void test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

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 models

  Use the compiler option for specifying a code model to take advantage of the different addressing modes available for the microcontroller and thereby also place functions in different parts of memory. To read more about code models, see *Code models and function memory attributes*, page 19.

- 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 13, and *Using function memory attributes*, page 20, 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 either `__no_init` or `const`. This is useful for individual data objects that must be located at a fixed address, for example variables with external requirements, or for populating any hardware tables similar to interrupt vector tables. Note that it is not possible to use this notation for absolute placement of individual functions.

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

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

At compile time, data and functions are placed in different segments as described in *Data segments*, page 28, and *Code segments*, page 34, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker command file, as described in *Placing segments in memory*, page 26.

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 the following combinations of keywords:

- `__no_init`
- `__no_init` and `const` (whithout initializers)
- `const` (with initializers).

To place a variable at an absolute address, the argument to the `@` operator and the `#pragma location` directive should be a literal number, representing the actual address.

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 */
```

In the following examples, there are two `const` declared objects, where the first is not initialized, and the second is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0x4002
__no_init const int beta;                /* OK */

const int gamma @ 0x4004 = 3;            /* OK */
```

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

The following example shows incorrect usage:

```
int delta @ 0x4006;                /* Error, neither */
                                   /* "__no_init" nor "const".*/
```

DATA AND FUNCTION PLACEMENT IN SEGMENTS

The following method can be used for placing data or functions in named segments other than default:

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. The named segment can either be a predefined segment, or a user-defined segment. The variables must be declared either `__no_init` or `const`. If declared `const`, they can have initializers.

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker command file using the `-Z` or the `-P` segment control directives.

Note: Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

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

Examples of placing variables in named segments

In the following three examples, a data object is placed in a user-defined segment. The segment will be allocated in default memory.

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

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

const int gamma @ "CONSTANTS" = 3;  /* OK */
```

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__data8 __no_init int alpha @ "NOINIT";/* Placed in data8*/
```

The following example shows incorrect usage:

```
int delta @ "NOINIT";               /* Error, neither */
                                    /* "__no_init" nor "const" */
```

Examples of placing functions in named segments

```
void f(void) @ "FUNCTIONS";

void g(void) @ "FUNCTIONS"
{
}

#pragma location="FUNCTIONS"
void h(void);
```

To override the default segment allocation, you can explicitly specify a memory attribute other than the default:

```
__banked void f(void) @ "FUNCTIONS";
```

Controlling compiler optimizations

The compiler performs many transformations on your application in order 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 there are some optimizations that 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.

In addition, you can 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 161, 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 it is also possible to make several source files into a compilation unit by using multi-file compilation. 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 115.

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 111.

OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. The following table lists the optimizations that are 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 Peephole optimization
Medium	Same as above Live-dead analysis and optimization Code hoisting Register content analysis and optimization Common subexpression elimination
High (Balanced)	Same as above Cross jumping Cross call (when optimizing for size) Loop unrolling Function inlining Code motion Type-based alias analysis

Table 25: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 85.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it will be 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 may 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. The following transformations can individually be disabled:

- 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 116.

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 118.

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 increases code size. The resulting code might also be difficult to debug.

The compiler decides which functions to inline. 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_inline*, page 117.

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 will take place using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers may reference the same memory location or not.

Type-based alias analysis is performed at optimization level **High**. For ISO/ANSI standard-conforming C application code, this optimization can reduce code size and execution time. However, non-standard-conforming 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 117.

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 code these pointers could overlap each other by being part of the same union. By using 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 116.

Writing efficient code

This section contains general programming hints on how to implement functions to make your applications robust, but at the same time facilitate compiler optimizations.

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 may 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. There are two main reasons why this is inefficient. 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. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also,

inlining may enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the `--no_inline` command line option; see *--no_inline*, page 117.

- Avoid using inline assembler. Instead, try writing the code in C, use intrinsic functions, or write a separate module in assembler language. For more details, see *Mixing C and assembler*, page 65.

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; in order to save stack space, you should instead pass them as pointers.

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. In addition, using the prototyped style will 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 a, int b)       /* definition */
{
  .....
}
```

Kernighan & Ritchie style

In K&R style—traditional pre-ISO/ANSI 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.

```
int test();                    /* old declaration */
int test(a,b)                  /* old definition */
char a;
int b;
{
  .....
}
```

INTEGER TYPES AND BIT NEGATION

There are situations when 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 may 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 the following 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. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 131.

A sequence that accesses a `volatile` declared variable must also not be interrupted. This can be achieved by using the `__monitor` keyword in interruptible code. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. This is true for all variables of all sizes. Accessing a small-sized variable can be an atomic operation, but this is not guaranteed and 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.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of S08 devices are included in the product installation. The header files are named `io`*`device`*`.h` and define the processor-specific special function registers (SFRs).

Each header file contains one section used by the compiler, and one section used by the assembler. The header files might in their turn include other header files provided by Freescale, typically named *`device`*`.h` and *`device`*`.inc` for the compiler and the assembler, respectively.

SFRs with bitfields are declared in the header files. This example is based on `mc9s08qe128.h`, which is included from `iomc9s08qe128.h`:

```
typedef unsigned char byte;

/*** PTAD - Port A Data Register; 0x00000000 ***/
typedef union {
  byte Byte;
  struct {
    byte PTAD0   :1;            /* Port A Data Register Bit 0 */
    byte PTAD1   :1;            /* Port A Data Register Bit 1 */
    byte PTAD2   :1;            /* Port A Data Register Bit 2 */
    byte PTAD3   :1;            /* Port A Data Register Bit 3 */
    byte PTAD4   :1;            /* Port A Data Register Bit 4 */
    byte PTAD5   :1;            /* Port A Data Register Bit 5 */
    byte PTAD6   :1;            /* Port A Data Register Bit 6 */
    byte PTAD7   :1;            /* Port A Data Register Bit 7 */
  } Bits;
} PTADSTR;

extern volatile PTADSTR _PTAD @0x00000000;
```

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:

```
/* whole register access */
_PTAD.Byte = 0x42;

/* bitfield access */
_PTAD.Bits.PTAD3 = 1;
_PTAD.Bits.PTAD4 = 0;
```

The header file also provides macros for convenience:

```
#define PTAD                        _PTAD.Byte
#define PTAD_PTAD0                  _PTAD.Bits.PTAD0
#define PTAD_PTAD1                  _PTAD.Bits.PTAD1
#define PTAD_PTAD2                  _PTAD.Bits.PTAD2
#define PTAD_PTAD3                  _PTAD.Bits.PTAD3
#define PTAD_PTAD4                  _PTAD.Bits.PTAD4
#define PTAD_PTAD5                  _PTAD.Bits.PTAD5
#define PTAD_PTAD6                  _PTAD.Bits.PTAD6
#define PTAD_PTAD7                  _PTAD.Bits.PTAD7

#define PTAD_PTAD0_MASK             1
#define PTAD_PTAD1_MASK             2
#define PTAD_PTAD2_MASK             4
#define PTAD_PTAD3_MASK             8
#define PTAD_PTAD4_MASK             16
#define PTAD_PTAD5_MASK             32
#define PTAD_PTAD6_MASK             64
#define PTAD_PTAD7_MASK             128
```

This allows you to write the same code like this:

```
/* whole register access */
PTAD = PTAD_PTAD1_MASK | PTAD_PTAD6_MASK;

/* bitfield access */
PTAD_PTAD3 = 1;
PTAD_PTAD4 = 0;
```

You can also use the header files as templates when you create new header files for other S08 devices. For details about the @ operator, see *Located data*, page 34.

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 separate segment, according to the specified memory keyword. See the chapter *Placing code and data* for more information.

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

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

For information about the __no_init keyword, see page 150. Note that to use this keyword, language extensions must be enabled; see *-e*, page 112. For information about the #pragma object_attribute, see page 161.

Part 2. Reference information

This part of the IAR C Compiler Reference Guide for S08 contains the following chapters:

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

External interface details

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

Invocation syntax

You can use the compiler either from the IDE or from the command line. Refer to the *IAR Embedded Workbench® IDE User Guide* for information about using the compiler from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccs08 [options] [sourcefile] [options]
```

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

```
iccs08 prog --debug
```

The source file can be a C file, typically with the filename extension `c`. 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.

PASSING OPTIONS

There are three different ways of passing options to the compiler:

- Directly from the command line

  Specify the options on the command line after the `iccs08` command, either before or after the source filename; see *Invocation syntax*, page 95.

- Via environment variables

  The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 96.
- Via a text file by using the `-f` option; see *-f*, page 113.

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

ENVIRONMENT VARIABLES

The following 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\s08\inc;c:\headers`
QCCS08	Specifies command line options; for example: `QCCS08=-lA asm.lst`

Table 26: Compiler environment variables

Include file search procedure

This is a detailed description of the compiler's `#include` file search procedure:

- If the name of the `#include` file is an absolute path, that file is opened.
- If the compiler encounters the name of an `#include` file in angle brackets, such as:

  ```
  #include <stdio.h>
  ```

  it searches the following directories for the file to include:

  1. The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 113.
  2. The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 96.

- 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 the following command for compilation:

```
iccs08 ..\src\src.c -I..\include -I..\debugconfig
```

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

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

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

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

Compiler output

The compiler can produce the following output:

- A linkable object file

  The object files produced by the compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `r78`.
- Optional list files

  Different types of list files can be specified using the compiler option `-l`, see *-l*, page 114. 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 `stderr` and displayed on the screen, as well as printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 98.
- 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 98.

- Size information

  Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` 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 will be retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

Error return codes

The compiler returns status information to the operating system that can be tested in a batch file.

The following command line error codes are supported:

Code	Description
0	Compilation successful, but there may have been warnings.
1	There were warnings and the option `--warnings_affect_exit_code` was used.
2	There were errors.
3	There were fatal errors making the compiler abort.
4	There were internal errors making the compiler abort.

Table 27: Error return codes

Diagnostics

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

MESSAGE FORMAT

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

`filename,linenumber  level[tag]: message`

with the following elements:

`filename`	The name of the source file in which the issue was encountered

linenumber	The line number at which the compiler detected the issue
level	The level of seriousness of the issue
tag	A unique tag that identifies the diagnostic message
message	An explanation, possibly several lines long

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

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construction that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 123.

Warning

A diagnostic message that is produced when the compiler finds a programming error or omission which is of concern, but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see page 119.

Error

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

Fatal error

A diagnostic message that is produced when the compiler has found a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the message has been 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 103, for a description of the compiler options that are available for setting severity levels.

See 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 has been a serious and unexpected failure due to a fault in the compiler. It is produced using the following form:

```
Internal error: message
```

where `message` is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Include enough information to reproduce the problem, typically:

- The product name
- The version number of the compiler, which can be seen in the header of the list files generated by the compiler
- Your license number
- The exact internal error message text
- The source file of the application that generated the internal error
- A list of the options that were used when the internal error occurred.

Compiler options

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

Options syntax

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

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 may 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 may 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 95.

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:

```
--misrac1998=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=filename
```

or

```
--diagnostics_tables filename
```

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 filename
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n filename
```

Rules for specifying a filename or directory as parameters

The following 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\`:

  ```
  iccs08 prog -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 has been specified with the option `-o`, in which case that name will be used. For example:

  ```
  iccs08 prog -l ..\listings\
  ```

  The produced list file will have the default name `..\listings\prog.lst`
- The *current directory* is specified with a period (`.`). For example:

  ```
  iccs08 prog -l .
  ```
- `/` can be used instead of `\` as the directory delimiter.
- By specifying `-`, input files and output files can be redirected to `stdin` and `stdout`, respectively. For example:

  ```
  iccs08 prog -l -
  ```

Additional rules

In addition, the following rules 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; the following example will create a list file called `-r`:

  ```
  iccs08 prog -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 may be repeated for each argument, for example:

  ```
  --diag_warning=Be0001
  --diag_warning=Be0002
  ```

Summary of compiler options

The following table summarizes the compiler command line options:

Command line option	Description
`--char_is_signed`	Treats `char` as signed
`--code_model`	Specifies the code model

Table 28: Compiler options summary

Command line option	Description
--core	Specifies the instruction set architecture
-D	Defines preprocessor symbols
--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	Determines the library configuration file
-e	Enables language extensions
--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
--library_module	Creates a library module
--mfc	Enables multi file compilation
--misrac1998	Enables error messages specific to MISRA C. See the *IAR Embedded Workbench® MISRA C Reference Guide*.
--misrac_verbose	Enables verbose logging of MISRA C checking. See the *IAR Embedded Workbench® MISRA C Reference Guide*.
--module_name	Sets the object module name
--no_code_motion	Disables code motion optimization
--no_cse	Disables common subexpression elimination
--no_inline	Disables function inlining

Table 28: Compiler options summary (Continued)

Command line option	Description
--no_path_in_file_macros	Removes the path from the return value of the symbols __FILE__ and __BASE_FILE__
--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
--omit_types	Excludes type information
--only_stdout	Uses standard output only
--output	Sets the object filename
--preinclude	Includes an include file before reading the source file
--preprocess	Generates preprocessor output
--public_equ	Defines a global named assembler label
-r	Generates debug information
--remarks	Enables remarks
--require_prototypes	Verifies that functions are declared before they are defined
--silent	Sets silent operation
--strict_ansi	Checks for strict compliance with ISO/ANSI C
--warnings_affect_exit_code	Warnings affects exit code
--warnings_are_errors	Warnings are treated as errors

Table 28: Compiler options summary (Continued)

Descriptions of options

The following section gives detailed reference information about each compiler option.

Note that if you use the IDE 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.

--char_is_signed

Syntax

`--char_is_signed`

Description

By default, the compiler interprets the `char` type as unsigned. Use this option to make the compiler interpret the `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. If you use this option, you may get type mismatch warnings from the linker, because the library uses `unsigned char`.

Project>Options>C Compiler>Language>Plain 'char' is

--code_model

Syntax

`--code_model={small|banked}`

Parameters

`small` (default)	Non-banked function calls
`banked`	Banked function calls

Description

Use this option to select the code model for which the code is to be generated. If you do not choose a code model option, 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 function memory attributes, page 19.

Project>Options>General Options>Target>Code model

--core

Syntax

`--core={V2|V4}`

Parameters

`V2` (default)	Instructions sets that do not include the code bank instructions `CALL` and `RTC`.
`V4`	Instructions sets that include the code bank instructions `CALL` and `RTC`.

Description

The compiler supports different devices of the S08 microcontroller family, with different instruction set architectures. Use this option to select the instruction set for which the code is to be generated.

To set related options, choose:

Project>Options>General Options>Target

-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`

In order to get the equivalence of:

```
#define FOO
```

specify the `=` sign but nothing after, for example:

```
-DFOO=
```

Project>Options>C Compiler>Preprocessor>Defined symbols

--debug, -r

Syntax

```
--debug
-r
```

Description

Use the `--debug` or `-r` option to make the compiler include information in the object modules required by C-SPY® and other symbolic debuggers.

Note: Including debug information will make the object files larger than otherwise.

Project>Options>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 102.

Description

Use this option to make the compiler list all source and header files opened by the compilation into a file with the default filename extension `i`.

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened source 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 uses makefile style. For each source 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 a source file. For example:

```
foo.r78: c:\iar\product\include\stdio.h
foo.r78: 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:

```
%.r78 : %.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 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 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 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 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 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 208.

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 from the compilation unit. This enhances interprocedural optimizations such as inlining, cross call, and cross jump by limiting their scope to public functions and variables that are actually used.

This option is only useful when *all* source files are compiled as one unit, which means that the `--mfc` compiler option is used.

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 115 and *Multi-file compilation units*, page 83.

Project>Options>C Compiler>Discard unused publics

--dlib_config

Syntax

```
--dlib_config filename
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 102.

Description

Each runtime library has a corresponding library configuration file. Use this option to specify the library configuration file for the compiler. Make sure that you specify a configuration file that corresponds to the library you are using.

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 `s08\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 40.

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 46.

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 enable them by using this option.

Note: The `-e` option and the `--strict_ansi` option cannot be used at the same time.

See also The chapter *Compiler extensions.*

Project>Options>C Compiler>Language>Allow IAR extensions

Note: By default, this option is enabled in the IDE.

--enable_multibytes

Syntax `--enable_multibytes`

Description By default, multibyte characters cannot be used in 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 style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

Project>Options>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 208.

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 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 may be used more than once on the command line.

See also

Include file search procedure, page 96.

Project>Options>C Compiler>Preprocessor>Additional include directories

-l

Syntax

`-l[a|A|b|B|c|C|D][N][H] {filename|directory}`

Parameters

`a`	Assembler list file
`A`	Assembler list file with 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 list file
`C` (default)	C list file with assembler source as comments
`D`	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 102.

Description

Use this option to generate an assembler or 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 Compiler>List

--library_module

Syntax

`--library_module`

Description

Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.

Project>Options>C Compiler>Output>Module type>Library Module

--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 makes interprocedural optimizations such as inlining, cross call, and cross jump possible.

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

```
iccs08 myfile1.c myfile2.c myfile3.c --mfc
```

See also

--discard_unused_publics, page 111, *-o, --output*, page 120, and *Multi-file compilation units*, page 83.

Project>Options>C Compiler>Multi-file compilation

--module_name

Syntax

`--module_name=`*`name`*

Parameters

name	An explicit object module name

Description

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.

This option is useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

Project>Options>C Compiler>Output>Object module name

--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 may be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>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 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 may be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C Compiler>Optimizations>Enable transformations>Common subexpression elimination

--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 and increases code size. The resulting code may also be difficult to debug.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed than for size.

Note: This option has no effect at optimization levels below High.

Project>Options>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 172.

This option is not available in the IDE.

--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 86.

Project>Options>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 the following:

```
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 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 may 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 Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax

`--no_warnings`

Description

By default, the compiler issues warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax

`--no_wrap_diagnostics`

Description

By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

-O

Syntax

`-O[n|l|m|h|hs|hz]`

Parameters

`n`	None* (Best debug support)
`l` (default)	Low*
`m`	Medium
`h`	High, balanced
`hs`	High, favoring speed
`hz`	High, favoring size

***The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.**

Description

Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only `-O` is used without any parameter, the optimization level High balanced is used.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

See also

Controlling compiler optimizations, page 83.

Project>Options>C Compiler>Optimizations

-o, --output

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 208.

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 `r78`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--omit_types

Syntax

```
--omit_types
```

Description

By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.

To set this option, use **Project>Options>C Compiler>Extra Options**.

--only_stdout

Syntax
```
--only_stdout
```

Description

Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).

This option is not available in the IDE.

--output, -o

Syntax
```
--output {filename|directory}
-o {filename|directory}
```

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 208.

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 `r78`. Use this option to explicitly specify a different output filename for the object code output.

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 102.

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 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 102.

Description Use this option to generate preprocessed output to a named file.

Project>Options>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 may be used more than once on the command line.

This option is not available in the IDE.

-r, --debug

Syntax `-r`
`--debug`

Description Use the `-r` or the `--debug` option to make the compiler include information in the object modules required by C-SPY and other symbolic debuggers.

Note: Including debug information will make the object files larger than otherwise.

Project>Options>C Compiler>Output>Generate debug information

--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 204.

Project>Options>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.

Note: This option only applies to functions in the C standard library.

Project>Options>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_ansi

Syntax `--strict_ansi`

Description By default, the compiler accepts a relaxed superset of ISO/ANSI C, see *Minor language extensions*, page 139. Use this option to ensure that the program conforms to the ISO/ANSI C standard.

Note: The `-e` option and the `--strict_ansi` option cannot be used at the same time.

Project>Options>C Compiler>Language>Language conformances>Strict ISO/ANSI

--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 296.

Project>Options>C Compiler>Diagnostics>Treat all warnings as errors

Data representation

This chapter describes the data types, pointers, and structure types supported by the compiler.

See the chapter *Efficient coding for embedded applications* for information about which data types 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.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

ALIGNMENT ON THE S08 MICROCONTROLLER

There are no alignment requirements for how the S08 microcontroller accesses memory.

Basic data types

The compiler supports both all ISO/ANSI C basic data types and some additional types.

INTEGER TYPES

The following 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

Table 29: Integer types

Signed variables are represented using the two's complement form.

Bool

If you have enabled language extensions, the `bool` type can 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, the `enum` constants and types can also be of the type `long` or `unsigned 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.

To use the `wchar_t` type, you must include the file `stddef.h` from the runtime library.

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the IAR C Compiler for S08, any integer type can be used as the base type when language extensions are enabled.

Bitfields in expressions will have the same data type as the integer base type.

By default, the compiler places bitfield members from the least significant to the most significant bit in the container type.

By using the directive `#pragma bitfields=reversed`, the bitfield members are placed from the most significant to the least significant bit.

FLOATING-POINT TYPES

In the IAR C Compiler for S08, floating-point values are represented in standard IEEE 754 format. The size of the floating-point types is 32 bits.

Exception flags for floating-point values are supported, and they are defined in the `fenv.h` file.

32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:

31	30 … 23	22 … 0
S	Exponent	Mantissa

The exponent is 8 bits, and the mantissa is 23 bits.

The value of the number is:

$(-1)^{S} * 2^{(Exponent-127)} * 1.Mantissa$

The range of the number is:

±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

The following 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 would have been 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.

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
__non_banked	0-0xFFFF	16 bits	Points to non-banked functions.
__banked	0-0xFFFFFF	24 bits	Points to banked functions.

Table 30: Function pointers

DATA POINTERS

Data pointers are 8 or 16 bits. The following data pointers are available:

Keyword	Pointer size	Index type	Pointer value range	Address range
`__data8`	8 bits	signed char		`0x0–0xFF`
`__data16`	16 bits	signed int		`0x0–0xFFFF`

Table 31: Data pointers

CASTING

Casts between pointers have the following 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 sign extension if the integer is `signed`, and by zero extension if it is `unsigned`
- Casting a *data pointer* to a larger data pointer is performed by zero extension
- Casting a *data pointer* to a smaller data pointer is performed by truncation
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a banked *function pointer* to a non-banked function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result.

Note: The only pointer-to-pointer cast that is guaranteed to provide a valid pointer as a result is casting from a `__data8` pointer to a `__data16` pointer.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the IAR C Compiler for S08, 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 Compiler for S08, 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 Compiler for S08, the size of intptr_t is 16 bits.

uintptr_t

uintptr_t is equivalent to intptr_t, with the exception that it is unsigned.

Structure types

The members of a struct are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

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;
```

The following diagram shows the layout in memory:

Figure 5: Structure layout

The alignment of the structure is 1 byte, and the size is 3 bytes.

Type qualifiers

According to the ISO/ANSI C standard, `volatile` and `const` are type qualifiers.

DECLARING OBJECTS VOLATILE

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 ISO/ANSI 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 that has been 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 underlaying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C Compiler for S08 are described below.

Rules for accesses

In the IAR C Compiler for S08, accesses to `volatile` declared objects are subject to the following rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The compiler adheres to these rules for all 8-bit data types and for direct accesses to statically allocated 16-bit data types.

For all other object and access types, only the rule that states that all accesses are preserved applies. Note in particular that this applies also to indirect 16-bit accesses (via a pointer).

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 data16 memory are allocated in ROM. For data8 memory, such objects are treated like static variables allocated in RAM and are initialized by the runtime system at startup.

Compiler extensions

This chapter gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically, the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C as well as 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.

You can find the extensions available as:

- C language extensions

  For a summary of available language extensions, see *C language extensions*, page 134. For reference information about the extended keywords, see the chapter *Extended keywords*.
- Pragma directives

  The `#pragma` directive is defined by the ISO/ANSI C standard 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 language extension. For a list of available pragma directives, see the chapter *Pragma directives*.
- Preprocessor extensions

  The preprocessor of the compiler adheres to the ISO/ANSI standard. In addition, the compiler also makes a number of 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 65. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

  The DLIB library provides most of the important C library definitions that apply to embedded systems. In addition, the library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 178.

Note: Any use of these extensions, except for the pragma directives, makes your application inconsistent with the ISO/ANSI C standard.

ENABLING LANGUAGE EXTENSIONS

In the IDE, language extensions are enabled by default.

For information about how to enable and disable language extensions from the command line, see the compiler options *-e*, page 112, and *--strict_ansi, page 124*.

C language extensions

This section gives a brief overview of the C language extensions available in the compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions have been grouped according to their expected usefulness. In short, this means:

- Important language extensions—extensions specifically tailored for efficient embedded programming, typically to meet memory restrictions
- Useful language extensions—features considered useful and typically taken from related standards, such as C99 and C++
- Minor language extensions, that is, the relaxation of some minor standards issues and also some useful but minor syntax extensions.

IMPORTANT LANGUAGE EXTENSIONS

The following language extensions are well suited for embedded systems programming:

- Memory attributes, type attributes, and object attributes

  For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.

- Placement at an absolute address or in a named segment

  The `@` operator or the directive `#pragma location` can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these primitives, see *Controlling data and function placement in memory, page 79*, and *location*, page 160.

- Alignment

  Each data type has its own alignment, for more details, see *Alignment*, page 125. If you want to change the alignment, the`#pragma data_alignment` directive is available. If you want to use 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 named 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 78.

- Bitfields and non-standard types

  In ISO/ANSI C, a bitfield must be of type `int` or `unsigned int`. Using IAR Systems language extensions, any integer type or enumeration may be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 127.

- Dedicated segment operators `__segment_begin` and `__segment_end`

  The syntax for these operators is:

  ```
  void * __segment_begin(segment)
  void * __segment_end(segment)
  ```

  These operators return the address of the first byte of the named *`segment`* and the first byte *after* the named *`segment`*, respectively. This can be useful if you have used the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment.

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

  In the following example, the type of the `__segment_begin` operator is `void __data8 *`.

  ```
  #pragma segment="MYSEGMENT" __data8
  ...
  segment_start_address = __segment_begin("MYSECTION");
  ```

  See also *segment*, page 164, and *location*, page 160.

USEFUL LANGUAGE EXTENSIONS

This section lists and briefly describes useful extensions, that is, useful features typically taken from related standards, such as C99 and C++:

- Inline functions

  The `#pragma inline` directive, alternatively the `inline` keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword `inline`. For more information, see *inline*, page 159.

- Mixing declarations and statements

  It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.

- Declaration in `for` loops

  It is possible to have a declaration in the initialization expression of a `for` loop, for example:

  ```
  for (int i = 0; i < 10; ++i)
  {...}
  ```

  This feature is part of the C99 standard and C++.

- The `bool` data type

  To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++.

- C++ style comments

  C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

  ```
  // The length of the bar, in centimeters.
  int length;
  ```

  This feature is copied from the C99 standard and C++.

Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict_ansi` is used. The `__asm` keyword is always available.

Note: Not all assembler directives or operators can be inserted using this keyword.

The syntax is:

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

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

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

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

For more information about inline assembler, see *Mixing C and assembler*, page 65.

Compound literals

To create compound literals you can use the following syntax:

```
/* Create a pointer to an anonymous array */
int *p = (int []) {1,2,3};

/* Create a pointer to an anonymous structX */
structX *px = &(structX) {5,6,7};
```

Note:

- A compound literal can be modified unless it is declared `const`
- This feature is part of the C99 standard.

Incomplete arrays at end of structs

The last element of a `struct` can be an incomplete array. This is useful for allocating a chunk of memory that contains both the structure and a fixed number of elements of the array. The number of elements can vary between allocations.

This feature is part of the C99 standard.

Note: The array cannot be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

Example

```
struct str
{
  char a;
  unsigned long b[];
};
```

```
struct str * GetAStr(int size)
{
  return malloc(sizeof(struct str) +
                sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
  s->b[10] = 0;
}
```

Hexadecimal floating-point constants

Floating-point constants can be given in hexadecimal style. The syntax is `0x`*`MANT`*`p{+|-}`*`EXP`*, where *`MANT`* is the mantissa in hexadecimal digits, including an optional `.` (decimal point), and *`EXP`* is the exponent with decimal digits, representing an exponent of 2. This feature is part of the C99 standard.

Examples

`0x1p0` is `1`

`0xA.8p2` is `10.5*2^2`

Designated initializers in structures and arrays

Any initialization of either a structure (`struct` or `union`) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as `.`*`elementname`* and for an array `[`*`constant index expression`*`]`.

Examples

The following definition shows a `struct` and its initialization using designators:

```
struct{
  int i;
  int j;
  int k;
  int l;
  short array[10];
} u = {
  .l = 6,           /* initialize l to 6 */
  .j = 6,           /* initialize j to 6 */
  8,                /* initialize k to 8 */
  .array[7] = 2,    /* initialize element 7 to 2 */
  .array[3] = 2,    /* initialize element 3 to 2 */
  5,                /* array[4] = 5 */
  .k = 4            /* reinitialize k to 4 */
};
```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```
union {
  int i;
  float f;
} y = {.f = 5.0};
```

To set the size of an array by initializing the last element, do like this:

```
char array[] = {[10] = 'a'};
```

MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards issues and also some useful but minor syntax extensions:

- Arrays of incomplete types

  An array may 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 IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- Missing semicolon at end of `struct` or `union` specifier

  A warning 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 ISO/ANSI C, some operators allow such things, while others do not allow them.

- Casting pointers to integers in static initializers

  In an initializer, a pointer constant value may 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 129.

- Taking the address of a register variable

  In ISO/ANSI 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.

- Duplicated size and sign specifiers

  Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error 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 will be 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 strict ISO/ANSI 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 ISO/ANSI mode, a warning is issued.

- A label preceding a `}`

  In ISO/ANSI 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. In the IAR C Compiler for S08, a warning is issued.

Note: 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

  ISO/ANSI 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. In the IAR C Compiler for S08, the following expression is allowed:

```
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 112.

Extended keywords

This chapter describes the extended keywords that support specific features of the S08 microcontroller and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

For information about the address ranges of the different memory areas, see the chapter *Segment reference*.

General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The compiler provides a set of attributes that can be used on functions or data objects to support specific features of the S08 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 147.

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 112 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 attributes* and *general type attributes*.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

- Available *function memory attributes*: `__banked`, and `__non_banked`
- Available *data memory attributes*: `__data8` and `__data16`

Data objects, functions, and destinations of pointers 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 may specify one (at most) memory attribute for each level of pointer indirection.

General type attributes

The following general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt`, `__monitor`, 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 131.

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 `__data8` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in data8 memory. The variables `k` and `l` behave in the same way:

```
__data8 int i, j;
int __data8 k, l;
```

Note that the attribute affects both identifiers.

The following declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__data8
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 15.

An easier way of specifying storage is to use type definitions. The following two declarations are equivalent:

```
typedef char __data8 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__data8 char b;
char __data8 *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 __data8 * p;`	The `int` object is located in data8 memory.
`int * __data8 p;`	The pointer is located in data8 memory.
`__data8 int * p;`	The pointer is located in data8 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);
```

The following 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 the following syntax:

```
int (__banked * fp) (double);
```

After this declaration, the function pointer `fp` points to banked memory.

An easier way of specifying storage is to use type definitions:

```
typedef __banked 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 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.

The following object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic` 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 79*.

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. The following 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

The following table summarizes the extended keywords:

Extended keyword	Description
__banked	Controls the storage of functions
__data8	Controls the storage of data objects
__data16	Controls the storage of data objects
__interrupt	Supports interrupt functions
__intrinsic	Reserved for compiler internal use only
__monitor	Supports atomic execution of a function
__no_init	Supports non-volatile memory
__non_banked	Controls the storage of functions
__noreturn	Informs the compiler that the declared function will not return
__root	Ensures that a function or variable is included in the object code even if unused
__task	Allows functions to exit without restoring registers

Table 32: Extended keywords summary

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

__banked

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 143.

Description

The __banked memory attribute overrides the default storage of functions given by the selected code model and places individual functions in banked memory. You can also

use the `__banked` attribute to create a pointer explicitly pointing to an object located in the banked memory.

Storage information

- Address range: `0-0xFFFFFF` (16 Mbytes)
- Maximum size: 16 Kbytes
- Pointer size: 3 bytes

Example

```
__banked void myfunction(void);
```

See also

Code models and function memory attributes, page 19.

__data8

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 143.

Description

The `__data8` memory attribute places individual variables and constants in data8 memory. You can also use the `__data8` attribute to create a pointer explicitly pointing to an object located in data8 memory.

Storage information

- Address range: `0-0xFF` (256 Kbytes)
- Maximum object size: 255 bytes
- Pointer size: 1 byte

Example

```
__data8 int x;
```

See also

Memory types, page 12.

__data16

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 143.

Description

The `__data16` memory attribute places individual variables and constants explicitly in data16 memory. You can also use the `__data16` attribute to create a pointer explicitly pointing to an object located in data16 memory.

Storage information

- Address range: `0-0xFFFF` (64 Kbytes)
- Maximum object size: 65535 bytes
- Pointer size: 2 bytes.

Example

```
__data16 int x;
```

See also

Memory types, page 12.

__interrupt

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 143.

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=Vswi
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 22, *vector*, page 165, *INTVEC*, page 189.

__intrinsic

Description

The `__intrinsic` keyword is reserved for compiler internal use only.

__monitor

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 143.

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 23. Read also about the intrinsic functions *__disable_interrupt*, page 168, *__enable_interrupt*, page 168, *__get_interrupt_state*, page 168, and *__set_interrupt_state*, page 169.

__no_init

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 146.

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];
```

__non_banked

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 143.

Description

The `__non_banked` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in non-banked memory. You can also use the `__non_banked` attribute to create a pointer explicitly pointing to an object located in non-banked memory.

Storage information

- Address range: `0-0xFFFF` (64 Kbytes)
- Maximum size: Equal to the largest available code memory range
- Pointer size: 2 bytes

Example

```
__non_banked void myfunction(void);
```

See also

Code models and function memory attributes, page 19.

__noreturn

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 146.

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 146.

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 modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide.*

__task

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 143.

Description

This keyword allows functions to exit without restoring registers and it is typically used for the `main` function.

By default, functions save the contents of used non-scratch registers (preserved registers) on the stack upon entry, and restore them at exit. Functions declared `__task` do not save any registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers needed by the calling function, you should only use `__task` on functions that do not return.

The function `main` may 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 may be declared `__task`.

Example

```
__task void my_handler(void);
```

Pragma directives

This chapter describes the pragma directives of the compiler.

The #pragma directive is defined by the ISO/ANSI C standard 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

The following table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
`bitfields`	Controls the order of bitfield members
`constseg`	Places constant variables in a named segment
`data_alignment`	Gives a variable a higher (more strict) alignment
`dataseg`	Places variables in a named segment
`diag_default`	Changes the severity level of diagnostic messages
`diag_error`	Changes the severity level of diagnostic messages
`diag_remark`	Changes the severity level of diagnostic messages
`diag_suppress`	Suppresses diagnostic messages
`diag_warning`	Changes the severity level of diagnostic messages
`include_alias`	Specifies an alias for an include file
`inline`	Inlines a function
`language`	Controls the IAR Systems language extensions
`location`	Specifies the absolute address of a variable, or places groups of functions or variables in named segments
`message`	Prints a message

Table 33: 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
segment	Declares a segment name to be used by intrinsic functions
type_attribute	Changes the declaration and definitions of a variable or function
vector	Specifies the vector of an interrupt function

Table 33: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.8.6)*, page 197 and.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

bitfields

Syntax

```
#pragma bitfields={reversed|default}
```

Parameters

reversed	Bitfield members are placed from the most significant bit to the least significant bit.
default	Bitfield members are placed from the least significant bit to the most significant bit.

Description

Use this pragma directive to control the order of bitfield members.

By default, the compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant

bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

See also

Bitfields, page 127.

constseg

Syntax

```
#pragma constseg=[__memoryattribute ]{SEGMENT_NAME|default}
```

Parameters

`__memoryattribute`	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
`SEGMENT_NAME`	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment for constants.

Description

Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma constseg=__data8 MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

data_alignment

Syntax

```
#pragma data_alignment=expression
```

Parameters

`expression`	A constant which must be a power of two (1, 2, 4, etc.).

Description

Use this pragma directive to give a variable a higher (more strict) alignment than it would otherwise have. It 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.

dataseg

Syntax

```
#pragma dataseg=[__memoryattribute ]{SEGMENT_NAME|default}
```

Parameters

__memoryattribute	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
SEGMENT_NAME	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment.

Description

Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared `__no_init`. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma dataseg=__data8 MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

diag_default

Syntax

```
#pragma diag_default=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description

Use this pragma directive to change the severity level back to default, or to the severity level defined on the command line by using 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 98.

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 98.

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 98.

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 98.

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 98.

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 96.

inline

Syntax
```
#pragma inline[=forced]
```

Parameters

`forced`	Disables the compiler's heuristics and forces inlining.

Description

Use this pragma directive 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 takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

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.

language

Syntax
```
#pragma language={extended|default}
```

Parameters

`extended`	Turns on the IAR Systems language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the language settings specified by compiler options.

Description

Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.

location

Syntax

```
#pragma location={address|NAME}
```

Parameters

address	The absolute address of the global or static variable for which you want an absolute location.
NAME	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

Description

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive.

Example

```
#pragma location=0x01
__no_init volatile char PTADD; /* PTADD is located at address
                                  0x01 */

#pragma location="FLASH"
int i;          /* i is located in segment FLASH */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
...
FLASH int i; /* i is placed in the FLASH segment */
```

See also

Controlling data and function placement in memory, page 79.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to `stdout`.

Description

Use this pragma directive to make the compiler print a message to `stdout` 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 146.

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 143.

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 no_inline
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 a double */
```

required

Syntax

```
#pragma required=symbol
```

Parameters

symbol	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.

Example

```
const char copyright[] = "Copyright by me";
...
#pragma required=copyright
int main()
{...}
```

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. In order 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 61.

__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 a double */
```

segment

Syntax

```
#pragma segment="NAME" [__memoryattribute]
```

Parameters

"*NAME*"	The name of the segment
__memoryattribute	An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.

Description

Use this pragma directive to define a segment name that can be used by the segment operators `__segment_begin` and `__segment_end`. All segment declarations for a specific segment must have the same memory type attribute.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

`void __memoryattribute *`.

Example

```
#pragma segment="MYSEGMENT" __data8
```

See also

Important language extensions, page 134. For more information about segments and segment parts, see the chapter *Placing code and data*.

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 143.

Description

Use this pragma directive to specify IAR-specific *type attribute*s, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute may 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 `__data8` is defined:

```
#pragma type_attribute=__data8
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__data8 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 function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example!

```
#pragma vector=0xFFFC
__interrupt void my_handler(void);
```

Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

The following table summarizes the intrinsic functions:

Intrinsic function	Description
`__BGND`	Inserts a `BGND` instruction
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__get_interrupt_state`	Returns the interrupt state
`__illegal_opcode`	Inserts an illegal operation code instruction
`__no_operation`	Inserts a `NOP` instruction
`__set_interrupt_state`	Restores the interrupt state
`__software_interrupt`	Inserts an `SWI` instruction
`__stop`	Inserts a `STOP` instruction
`__wait_for_interrupt`	Inserts a `WAIT` instruction

Table 34: Intrinsic functions summary

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

__BGND

Syntax

```
void __BGND(void)
```

Description

Enters active background debug mode by inserting a `BGND` instruction.

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts by inserting the `SEI` instruction.

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts by inserting the `CLI` 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

```
__istate_t s = __get_interrupt_state();
__disable_interrupt();

  /* Do something */

__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`.

__illegal_opcode

Syntax `void __illegal_opcode(void)`

Description Inserts an illegal operation code.

__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 168.

__software_interrupt

Syntax `void __software_interrupt(void)`

Description Inserts an `SWI` instruction.

__stop

Syntax `void __stop(void)`

Description Stops the processing by inserting a `STOP` instruction.

__wait_for_interrupt

Syntax `void __wait_for_interrupt(void)`

Description Inserts a `WAIT` 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 Compiler for S08 adheres to the ISO/ANSI standard. The compiler also makes the following 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 172.

- 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 107.

- 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 173.

- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 122.

Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 196.

Descriptions of predefined preprocessor symbols

The following 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 172*, and *–no_path_in_file_macros*, page 117.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the compiler currently in use.
`__CODE_MODEL__`	An integer that identifies the code model in use. The symbol reflects the `--code_model` option and is defined to `__CODE_MODEL_BANKED__` or `__CODE_MODEL_SMALL__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.
`__CORE__`	An integer that identifies the instruction set architecture you are compiling for. The symbol reflects the `--core` option and is defined to `__CORE_V2__` or `__CORE_V4__`. These symbolic names can be used when testing the `__CORE__` symbol.
`__DATE__`	A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2005`". *
`__FILE__`	A string that identifies the name of the file being compiled, which can be the base source file as well as any included header file. See also *__BASE_FILE__*, page 172, and *–no_path_in_file_macros*, page 117.*
`__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 112. See also *__PRETTY_FUNCTION__*, page 173.
`__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 112. See also *__PRETTY_FUNCTION__*, page 173.
`__IAR_SYSTEMS_ICC__`	An integer that identifies the IAR compiler platform. The current value is 7. 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.
`__ICCS08__`	An integer that is set to `1` when the code is compiled with the IAR C Compiler for S08, and otherwise to `0`.

Table 35: Predefined symbols

Predefined symbol	Identifies
`__LINE__`	An integer that identifies the current source line number of the file being compiled, which can be the base source file as well as any included header file.*
`__LITTLE_ENDIAN__`	An integer that identifies the byte order of the microcontroller. For the S08 microcontroller families, the value of this symbol is defined to `0` (`FALSE`), which means that the 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 112. See also *__func__*, page 172.
`__STDC__`	An integer that is set to `1`, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to ISO/ANSI C.*
`__STDC_VERSION__`	An integer that identifies the version of ISO/ANSI C standard in use. The symbols expands to `199409L`.*
`__SUBVERSION__`	An integer that identifies the third position of the compiler version number, for example the 3 in 1.10.3.
`__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 the following 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 35: Predefined symbols (Continued)

* **This symbol is required by the ISO/ANSI standard.**

Descriptions of miscellaneous preprocessor extensions

The following section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI 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 have written any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

_Pragma()

Syntax

`_Pragma("string")`

where *`string`* follows the syntax of the corresponding pragma directive.

Description

This preprocessor operator is part of the C99 standard and can be used, for example, in defines and is equivalent to the `#pragma` directive.

Note: The `-e` option—enable language extensions—does not have to be specified.

Example

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=none")
#else
  #define NOOPT
#endif
```

See also

See the chapter *Pragma directives*.

#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 ISO/ANSI standard `#error` directive is used.

__VA_ARGS__

Syntax

```
#define P(...)         __VA_ARGS__
#define P(x,y,...)     x + y + __VA_ARGS__
```

`__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

Description

Variadic macros are the preprocessor macro equivalents of `printf` style functions. `__VA_ARGS__` is part of the C99 standard.

Example

```
#if DEBUG
  #define DEBUG_TRACE(S,...) printf(S,__VA_ARGS__)
#else
  #define DEBUG_TRACE(S,...)
#endif
/* Place your own code here */
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

Library functions

This chapter gives an introduction to the 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.

Introduction

The compiler is delivered with the IAR DLIB Library, a complete ISO/ANSI C library. 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.

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 a number of 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 settings for project configuration*, page 5. 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 as they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`
- 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 in some way. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

In addition, some functions 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 has been 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 library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the C programming language. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard 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 S08 features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, partly taken from the C99 standard, see *Added C functionality*, page 180.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Compiler extensions*.

The following table lists the C header files:

Header file	Usage
`assert.h`	Enforcing assertions when functions execute
`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
`time.h`	Converting between various time and date formats
`wchar.h`	Support for wide characters
`wctype.h`	Classifying wide characters

Table 36: Traditional standard C header files—DLIB

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality, partly taken from the C99 standard.

The following include files provide these features:

- `ctype.h`
- `fenv.h`
- `inttypes.h`
- `math.h`
- `stdbool.h`
- `stdint.h`
- `stdio.h`
- `stdlib.h`
- `wchar.h`
- `wctype.h`

ctype.h

In `ctype.h`, the C99 function `isblank` is defined.

fenv.h

In `fenv.h`, support for exception flags for floating-point according to the C99 standard is defined.

inttypes.h

This include file defines the formatters for all types defined in `stdint.h` to be used by the functions `printf`, `scanf`, and all their variants.

math.h

In `math.h` all functions exist in a `float` variant and a `long double` variant, suffixed by `f` and `l` respectively. For example, `sinf` and `sinl`.

The following C99 macro symbols are defined:

HUGE_VALF, HUGE_VALL, INFINITY, NAN, FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL, FP_ZERO, MATH_ERRNO, MATH_ERREXCEPT, math_errhandling.

The following C99 macro functions are defined:

fpclassify, signbit, isfinite, isinf, isnan, isnormal, isgreater, isless, islessequal, islessgreater, isunordered.

The following C99 type definitions are added:

float_t, double_t.

stdbool.h

This include file makes the bool type available if the **Allow IAR extensions** (-e) option is used.

stdint.h

This include file provides integer characteristics.

stdio.h

In stdio.h, the following C99 functions are defined:

vscanf, vfscanf, vsscanf, vsnprintf, snprintf

The functions printf, scanf, and all their variants have added functionality from the C99 standard. For reference information about these functions, see the library reference available from the **Help** menu.

The following functions providing I/O functionality for libraries built without FILE support are definded:

__write_array	Corresponds to fwrite on stdout.
__ungetchar	Corresponds to ungetc on stdout.
__gets	Corresponds to fgets on stdin.

stdlib.h

In stdlib.h, the following C99 functions are defined:

_Exit, llabs, lldiv, strtoll, strtoull, atoll, strtof, strtold.

The function strtod has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The __qsort function is implemented using a bubble sort algorithm.

wchar.h

In wchar.h, the following C99 functions are defined:

vfwscanf, vswscanf, vwscanf, wcstof, wcstolb.

wctype.h

In wctype.h, the C99 function iswblank is defined.

Segment reference

The compiler places code and data into named segments which are referred to by the linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For more information about segments, see the chapter *Placing code and data*.

Summary of segments

The table below lists the segments that are available in the compiler:

Segment	Description
BANKED	Holds the banked program code.
CHECKSUM	Holds the checksum generated by the linker.
CODE	Holds the non-banked program code.
CSTACK	Holds the stack used by C programs.
DATA8_AC	Holds __data8 located constant data.
DATA8_AN	Holds __data8 located uninitialized data.
DATA8_I	Holds __data8 static and global initialized variables and constants.
DATA8_ID	Holds initial values for __data8 static and global variables and constants in DATA8_I.
DATA8_N	Holds __no_init __data8 static and global variables.
DATA8_Z	Holds zero-initialized __data8 static and global variables.
DATA16_AC	Holds __data16 located constant data.
DATA16_AN	Holds __data16 located uninitialized data.
DATA16_C	Holds __data16 constant data.
DATA16_I	Holds __data16 static and global initialized variables.
DATA16_ID	Holds initial values for __data16 static and global variables in DATA16_I.
DATA16_N	Holds __no_init __data16 static and global variables.
DATA16_Z	Holds zero-initialized __data16 static and global variables.
HEAP	Holds the heap data used by malloc and free.

Table 37: Segment summary

Segment	Description
INTVEC	Contains the reset and interrupt vectors.
RCODE	Holds assembler support routines used for system initialization and termination.

Table 37: Segment summary (Continued)

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by using the segment placement linker directives `-Z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—`CODE`, `CONST`, or `DATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 7, *XLINK segment memory types*, page 26.

For information about the `-Z` and the `-P` directives, see the *IAR Linker and Library Tools Reference Guide.*

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 26.

For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

BANKED

Description — Holds banked program code, except the code for system initialization.

Segment memory type — `CODE`

Memory placement — This segment can be placed in banked or non-banked code memory.

Access type — Read-only

CHECKSUM

Description — Holds the checksum generated by the linker.

Segment memory type — `CONST`

Memory placement — This segment can be placed anywhere in non-banked ROM memory.

Access type — Read-only

CODE

Description — Holds non-banked program code, except the code for system initialization.

Segment memory type — `CODE`

Memory placement — This segment must be placed in non-banked code memory.

Access type — Read-only

CSTACK

Description — Holds the internal data stack.

Segment memory type — `DATA`

Memory placement — This segment must be placed in non-banked RAM memory.

Access type — Read/write

See also — *The stack*, page 32.

DATA8_AC

Description — Holds `__data8` located constant data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA8_AN

Description — Holds `__no_init __data8` located data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA8_I

Description	Holds `__data8` static and global initialized variables and constants initialized by copying from the segment `DATA8_ID` at application startup.
Segment memory type	`DATA`
Memory placement	This segment must be placed in the first 256 bytes of memory.
Access type	Read/write

DATA8_ID

Description	Holds initial values for `__data8` static and global variables and constants in the `DATA8_I` segment. These values are copied from `DATA8_ID` to `DATA8_I` at application startup.
Segment memory type	`CONST`
Memory placement	This segment must be placed in the first 64 Kbytes of memory.
Access type	Read-only

DATA8_N

Description	Holds static and global `__no_init __data8` variables.
Segment memory type	`DATA`
Memory placement	This segment must be placed in the first 256 bytes of memory.
Access type	Read/write

DATA8_Z

Description	Holds zero-initialized `__data8` static and global variables. The contents of this segment is declared by the system startup code.
Segment memory type	`DATA`
Memory placement	This segment must be placed in the first 256 bytes of memory.

Access type Read/write

DATA16_AC

Description Holds `__data16` located constant data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA16_AN

Description Holds `__no_init __data16` located data.

Located means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA16_C

Description Holds `__data16` constant data.

Segment memory type `CONST`

Memory placement This segment must be placed in non-banked memory.

Access type Read-only

DATA16_I

Description Holds `__data16` static and global initialized variables initialized by copying from the segment `DATA16_ID` at application startup.

Segment memory type `DATA`

Memory placement This segment must be placed in non-banked RAM memory.

Access type Read/write

DATA16_ID

Description	Holds initial values for `__data16` static and global variables in the `DATA16_I` segment. These values are copied from `DATA16_ID` to `DATA16_I` at application startup.
Segment memory type	`CONST`
Memory placement	This segment must be placed in the first 64 Kbytes of memory.
Access type	Read-only

DATA16_N

Description	Holds static and global `__no_init __data16` variables.
Segment memory type	`DATA`
Memory placement	This segment must be placed in non-banked RAM memory.
Access type	Read/write

DATA16_Z

Description	Holds zero-initialized `__data16` static and global variables. The contents of this segment is declared by the system startup code.
Segment memory type	`DATA`
Memory placement	This segment must be placed in non-banked RAM memory.
Access type	Read/write

HEAP

Description	Holds the heap used for dynamically allocated data, in other words data allocated by `malloc` and `free`.
Segment memory type	`DATA`
Memory placement	This segment must be placed in non-banked RAM memory.

Access type	Read/write

See also *The heap*, page 33.

INTVEC

Description

Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword in combination with the `#pragma vector` directive.

RCODE

Description

Holds assembler support routines used for system initialization and termination.

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.

Segment memory type `CODE`

Memory placement This segment must be placed in non-banked code memory.

Access type Read-only

Implementation-defined behavior

This chapter describes how the compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1,* and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The compiler adheres to a freestanding implementation of the ISO standard for the C programming language. 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 ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

`filename,linenumber level[tag]: message`

where *`filename`* is the name of the source file in which the error was encountered, *`linenumber`* is the line number at which the compiler detected the error, *`level`* is the level of seriousness of the message (remark, warning, error, or fatal error), *`tag`* is a unique tag that identifies the message, and *`message`* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.2.1)

The function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the DLIB runtime environment, see *Customizing system initialization*, page 50.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The DLIB library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 54.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C Compiler—is the 'C' locale. If you use the command line option `--enable_multibytes`, the DLIB library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 54.

Range of 'plain' char (6.2.1.1)

A 'plain' `char` has the same range as an `unsigned char`.

INTEGERS

Range of integer values (6.1.2.5)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types*, page 126, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (`s`), a biased exponent (`e`), and a mantissa (`m`).

See *Floating-point types*, page 127, for information about the ranges and sizes for the different floating-point types: `float` and `double`.

Converting integer values to floating-point values (6.2.1.3)

When an integral number is cast to a floating-point value that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

Demoting floating-point values (6.2.1.4)

When a floating-point value is converted to a floating-point value of narrower type that cannot exactly represent the value, the value is rounded (up or down) to the nearest suitable value.

ARRAYS AND POINTERS

size_t (6.3.3.4, 7.1.1)

See *size_t*, page 129, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 129, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 129, for information about the `ptrdiff_t`.

REGISTERS

Honoring the register keyword (6.5.1)

User requests for register variables are not honored.

STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

Improper access to a union (6.3.2.3)

If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

Padding and alignment of structure members (6.5.2.1)

See the section *Basic data types*, page 126, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include`

directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect:

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
memory
module_name
no_pch
```

```
once
__printf_args
public_equ
__scanf_args
section
STDC
system_include
warnings
```

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB LIBRARY FUNCTIONS

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to `0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

filename:*linenr expression* `-- assertion failed`

when the parameter evaluates to zero.

Domain errors (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns `NaN`; `errno` is set to `EDOM`.

signal() (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 57.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

No null characters are appended to data written to binary streams.

Files (7.9.3)

Whether 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 54.

remove() (7.9.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 54.

rename() (7.9.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 54.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

Message generated by perror() (7.9.10.4)

The generated message is:

`usersuppliedprefix``:`*`errormessage`*

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 57.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 57.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

<table>
<tr><th>Argument</th><th>Message</th></tr>
<tr><td>EZERO</td><td>no error</td></tr>
<tr><td>EDOM</td><td>domain error</td></tr>
<tr><td>ERANGE</td><td>range error</td></tr>
<tr><td>EFPOS</td><td>file positioning error</td></tr>
<tr><td>EILSEQ</td><td>multi-byte encoding error</td></tr>
<tr><td><0 || >99</td><td>unknown error</td></tr>
<tr><td>all others</td><td>error nnn</td></tr>
</table>

Table 38: Message returned by strerror()—IAR DLIB library

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 58.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 58.

A

abort
implementation-defined behavior (DLIB) 200
system termination (DLIB) 49
absolute location
data, placing at (@) 80
language support for 134
#pragma location 160
addressing. *See* memory types and code models
alignment 125
forcing stricter (#pragma data_alignment) 155
of an object (__ALIGNOF__) 135
of data types. 125
alignment (pragma directive) 197
__ALIGNOF__ (operator) 135
anonymous structures 78
anonymous symbols, creating 137
application
building, overview of 4
startup and termination (DLIB) 47
architecture, S08 11
ARGFRAME (assembler directive) 74
arrays
designated initializers in 138
hints about index type 77
implementation-defined behavior. 195
incomplete at end of structs 137
non-lvalue 140
of incomplete types 139
single-value initialization. 141
asm, __asm (language extension) 136
assembler code
calling from C 68
inline 67
assembler directives, in inline assembler code 67
assembler instructions, inserting inline 67
assembler labels, making public (--public_equ) 122
assembler language interface 65
calling convention. *See* assembler code
assembler list file, generating 114
assembler output file 69
assembler, inline 136
asserts 58
implementation-defined behavior of, (DLIB). 198
including in application 174
assert.h (DLIB header file) 179
atoll, C99 extension 181
atomic operations 23
__monitor 149
attributes
object 146
type 143
auto variables 16
at function entrance 71
programming hints for efficient code 87
using in inline assembler code 67

B

__banked (extended keyword) 147
Banked code model. 19
banked code, using PPAGE register 3
banked function calls 72
indirect. 21
banked functions
address syntax 72
placing 72
banked systems, coding hints 21
__banked (extended keyword) 20
__banked (function pointer) 128
BANKED (segment). 184
banking 20, 72
Barr, Michael xxi
baseaddr (pragma directive) 197
__BASE_FILE__ (predefined symbol). 172
__BGND (intrinsic function) 168

BGND (assembler instruction) 168
binary streams (DLIB) 199
bit negation 89
bitfields
data representation of 127
implementation-defined behavior of 195
non-standard types in 135
specifying order of members (#pragma bitfields) 154
bitfields (pragma directive) 154
bold style, in this guide xxii
bool (data type) 126
adding support for in DLIB 179
making available in C code 181
bubble sort function, defined in stdlib.h 182
building_runtime (pragma directive) 197
__BUILD_NUMBER__ (predefined symbol) 172
byte order, identifying (__LITTLE_ENDIAN__) 173

C

C calling convention. *See* calling convention
C header files 179
call frame information 75
in assembler list file 69
in assembler list file (-lA) 114
call stack 75
callee-save registers, stored on stack 16
calling convention 70
calloc (library function) 17
See also heap
implementation-defined behavior of (DLIB) 200
can_instantiate (pragma directive) 197
casting
between pointer types 15
of pointers and integers 129
cexit (system termination code)
in DLIB 47
placement in segment 34
CFI (assembler directive) 75
char (data type) 126
changing default representation (--char_is_signed) . . . 106
signed and unsigned 127
characters, implementation-defined behavior of 192
character-based I/O
in DLIB 51
overriding in runtime library 44
--char_is_signed (compiler option) 106
CHECKSUM (segment) 184
CLI (assembler instruction) 168
clock (DLIB library function),
implementation-defined behavior of 201
clock.c 58
__close (DLIB library function) 54
code
banked 20, 72
execution of 6
interruption of execution 22
that must be non-banked 21
verifying linked result 35
code models 19
configuration 6
identifying (__CODE_MODEL__) 172
specifying on command line (--code_model) 106
code motion (compiler transformation) 86
disabling (--no_code_motion) 116
code paging, *see* banking
code segments, used for placement 34
CODE (segment) 185
codeseg (pragma directive) 197
__CODE_MODEL__ (predefined symbol) 172
--code_model (compiler option) 106
__code_model (runtime model attribute) 62
command line options
part of compiler invocation syntax 95
passing 95
See also compiler options
command prompt icon, in this guide xxiii
comments
after preprocessor directives 141

C++ style, using in C code . . . 136
common block (call frame information) . . . 75
common subexpr elimination (compiler transformation) . . 85
disabling (--no_cse) . . . 116
compilation date
exact time of (__TIME__) . . . 173
identifying (__DATE__) . . . 172
compiler
environment variables . . . 96
invocation syntax . . . 95
output from . . . 97
compiler listing, generating (-l). . . . 114
compiler object file
including debug information in (--debug, -r) . . . 107
compiler optimization levels . . . 84
compiler options . . . 101
passing to compiler . . . 95
reading from file (-f) . . . 113
setting . . . 101
specifying parameters . . . 103
summary . . . 103
syntax. . . . 101
typographic convention . . . xxii
compiler platform, identifying . . . 172
compiler subversion number . . . 173
compiler transformations . . . 83
compiler version number . . . 173
compiling
from the command line . . . 4
syntax. . . . 95
compound literals . . . 137
computer style, typographic convention . . . xxii
configuration
basic project settings . . . 5
__low_level_init . . . 50
configuration symbols, in library configuration files. . . . 46
consistency, module . . . 61
constants, placing in named segment . . . 155
constseg (pragma directive) . . . 155
const, declaring objects . . . 132
contents, of this guide . . . xx
conventions, used in this guide . . . xxii
copyright notice . . . ii
__CORE__ (predefined symbol). . . . 172
--core (compiler option) . . . 106
cross call (compiler transformation) . . . 87
banking considerations. . . . 22
cspy_support (pragma directive) . . . 197
CSTACK (segment) . . . 185
example . . . 32
See also stack
cstartup (system startup code) . . . 34
customizing . . . 50
overriding in runtime library . . . 44
cstartup.s78 . . . 47
ctype.h (library header file) . . . 179
added C functionality . . . 180
C++-style comments . . . 136
C-SPY, low-level interface to . . . 59
C_INCLUDE (environment variable) . . . 96
C99 standard, added functionality from . . . 180

D

-D (compiler option) . . . 107
data
alignment of . . . 125
placing . . . 79, 156, 183
at absolute location . . . 80
representation of . . . 125
storage . . . 11
verifying linked result . . . 35
data block (call frame information). . . . 75
data memory attributes, using . . . 13
data pointers . . . 129
data segments . . . 28
data types . . . 126
avoiding signed . . . 77

floating point 127
integers 126
dataseg (pragma directive) 156
data_alignment (pragma directive) 155
__data8 (extended keyword) 129, 148
data8 (memory type) 12
DATA8_AC (segment) 185
DATA8_AN (segment) 185
DATA8_I (segment) 186
DATA8_ID (segment) 186
DATA8_N (segment) 186
DATA8_Z (segment) 186
__data16 (extended keyword) 148
data16 (memory type) 13
DATA16_AC (segment) 187
DATA16_AN (segment) 187
DATA16_C (segment) 187
DATA16_I (segment) 187
DATA16_ID (segment) 188
DATA16_N (segment) 188
DATA16_Z (segment) 188
__DATE__ (predefined symbol) 172
date (library function), configuring support for 58
--debug (compiler option) 107
debug information, including in object file 107, 122
declarations
empty 141
in for loops 136
Kernighan & Ritchie 89
of functions 70
declarations and statements, mixing 136
declarators, implementation-defined behavior 196
default function type, overriding 20
define_type_info (pragma directive) 197
--dependencies (compiler option) 108
diagnostic messages 98
classifying as errors 109
classifying as remarks 109
classifying as warnings 110
disabling warnings 119
disabling wrapping of 119
enabling remarks 123
listing all used 110
suppressing 110
--diagnostics_tables (compiler option) 110
diag_default (pragma directive) 156
--diag_error (compiler option) 109
diag_error (pragma directive) 157
--diag_remark (compiler option) 109
diag_remark (pragma directive) 157
--diag_suppress (compiler option) 110
diag_suppress (pragma directive) 157
--diag_warning (compiler option) 110
diag_warning (pragma directive) 158
directives
function for static overlay 74
pragma 8, 153
directory, specifying as parameter 102
__disable_interrupt (intrinsic function) 168
--discard_unused_publics (compiler option) 111
disclaimer ii
DLIB 6, 178
building customized library 38
configurations 39
configuring 38, 111
debug support 39
reference information. *See* the online help system 177
runtime environment 37
--dlib_config (compiler option) 111
Dlib_defaults.h (library configuration file) 46
dls08*libname*.h 46
document conventions xxii
documentation, library 177
domain errors, implementation-defined behavior 198
double_t, C99 extension 181
do_not_instantiate (pragma directive) 197
dynamic initialization 47
dynamic memory 17

E

-e (compiler option) 112
early_initialization (pragma directive) 197
edition, of this guide ii
embedded systems, IAR special support for 8
__enable_interrupt (intrinsic function) 168
--enable_multibytes (compiler option) 112
entry label, program 48
enumerations, implementation-defined behavior. 195
enums
  data representation 126
  forward declarations of 139
environment
  implementation-defined behavior. 192
  runtime (DLIB) 37
environment variables
  C_INCLUDE. 96
  QCCS08. 96
EQU (assembler directive) 122
errno.h (library header file). 179
error messages 99
  classifying 109
error return codes 98
--error_limit (compiler option) 112
exception flags, for floating-point values 127
exception vectors 35
_Exit (library function). 49
exit (library function) 49
  implementation-defined behavior. 200
_exit (library function) 49
__exit (library function) 49
extended command line file 113
extended keywords 143
  enabling (-e). 112
  overview 8
  summary 147
  syntax. 13
    object attributes. 146
    type attributes on data objects 144
    type attributes on data pointers 145
    type attributes on function pointers 146
    type attributes on functions 145

F

-f (compiler option). 113
fatal error messages 99
fenv.h (library header file). 179
  additional C functionality. 180
fgetpos (library function), implementation-defined behavior 200
__FILE__ (predefined symbol). 172
file dependencies, tracking 108
file paths, specifying for #include files 113
filename, specifying as parameter. 102
floating-point constants, hexadecimal notation of. 138
floating-point format. 127
  hints 77
  implementation-defined behavior. 194
  special cases. 128
  32-bits 127
float.h (library header file) 179
float_t, C99 extension. 181
fmod (library function), implementation-defined behavior 198
for loops, declarations in. 136
formats
  floating-point values 127
  standard IEEE (floating point) 127
fpclassify, C99 extension 181
FP_INFINITE, C99 extension 181
FP_NAN, C99 extension. 181
FP_NORMAL, C99 extension 181
FP_SUBNORMAL, C99 extension 181
FP_ZERO, C99 extension. 181
fragmentation, of heap memory 18
free (library function). *See also* heap 17

ftell (library function), implementation-defined behavior . 200
Full DLIB (library configuration) 39
__func__ (predefined symbol) 142, 172
FUNCALL (assembler directive) 74
__FUNCTION__ (predefined symbol) 142, 172
function calls
  banked vs. non-banked. 21
  calling convention 70
function declarations, Kernighan & Ritchie 89
function directives for static overlay 74
function inlining (compiler transformation) 86
  disabling (--no_inline) 117
function memory attributes 20
function pointers 128
function prototypes 88
  enforcing 123
function return addresses 72
function type information, omitting in object output. 120
FUNCTION (assembler directive) 74
function (pragma directive). 197
functions 19
  banked
    address syntax. 72
    placing 72
  declared without attribute, placement. 35
  declaring 70, 88
  inlining. 86–87, 136, 159
  interrupt 22–23
  intrinsic 65, 88
  monitor 23
  non-banked, placing. 73
  omitting type info 120
  parameters 71
  placing in memory 79, 81
  recursive
    avoiding 88
    storing data on stack 17
  reentrancy (DLIB) 178
  related extensions. 19
  return values from 72
  special function types. 22
  verifying linked result 35

G

getenv (library function), configuring support for 57
getzone (library function), configuring support for 58
getzone.c 58
__get_interrupt_state (intrinsic function) 168
global variables, initialization of 31
guidelines, reading xix

H

Harbison, Samuel P. xxi
hardware support in compiler 38
hdrstop (pragma directive) 197
header files
  C 179
  library 177
  special function registers 90
  Dlib_defaults.h. 46
  dls08*libname*.h. 46
  intrinsics.h 167
  stdbool.h 126, 179
  stddef.h 127
--header_context (compiler option). 113
heap
  dynamic memory 17
  segment for 33
  storing data 12
heap segment
  HEAP. 188
  placing 34
heap size
  and standard I/O. 34
  changing default. 33
HEAP (segment). 33, 188

hints
banked systems . . . 21
optimization . . . 87
HUGE_VALF, C99 extension . . . 181
HUGE_VALL, C99 extension . . . 181

I

-I (compiler option) . . . 113
IAR Command Line Build Utility . . . 46
IAR Systems Technical Support . . . 100
iarbuild.exe (utility) . . . 46
__IAR_SYSTEMS_ICC__ (predefined symbol) . . . 172
__ICCS08__ (predefined symbol) . . . 172
icons, in this guide . . . xxiii
identifiers, implementation-defined behavior . . . 192
IEEE format, floating-point values . . . 127
__illegal_opcode (intrinsic function) . . . 169
implementation-defined behavior . . . 191
important_typedef (pragma directive) . . . 197
include files
including before source files . . . 121
specifying . . . 96
include_alias (pragma directive) . . . 158
indirect banked calls . . . 21
infinity . . . 128
INFINITY, C99 extension . . . 181
initialization
dynamic . . . 47
single-value . . . 141
initialized data segments . . . 31
initializers, static . . . 140
inline assembler . . . 67, 136
avoiding . . . 88
See also assembler language interface
inline functions . . . 136
in compiler . . . 86
inline (pragma directive) . . . 159
instantiate (pragma directive) . . . 197
instruction set architecture
identifying . . . 172
specifying on command line (--core) . . . 106
integer characteristics, adding . . . 181
integers . . . 126
casting . . . 129
implementation-defined behavior . . . 194
intptr_t . . . 130
ptrdiff_t . . . 129
size_t . . . 129
uintptr_t . . . 130
integral promotion . . . 89
internal error . . . 100
__interrupt (extended keyword) . . . 22, 149
using in pragma directives . . . 165
interrupt functions . . . 22
cannot be banked . . . 21
placement in memory . . . 35
interrupt state, restoring . . . 169
interrupt vector table . . . 23
INTVEC segment . . . 189
interrupt vector, specifying with pragma directive . . . 165
interrupts
disabling . . . 149
during function execution . . . 23
processor state . . . 16
intptr_t (integer type) . . . 130
__intrinsic (extended keyword) . . . 149
intrinsic functions . . . 88
overview . . . 65
summary . . . 167
intrinsics.h (header file) . . . 167
inttypes.h (library header file) . . . 179
inttypes.h, added C functionality . . . 180
INTVEC (segment) . . . 189
invocation syntax . . . 95
isblank, C99 extension . . . 180
isfinite, C99 extension . . . 181
isgreater, C99 extension . . . 181

isinf, C99 extension 181
islessequal, C99 extension 181
islessgreater, C99 extension 181
isless, C99 extension. 181
isnan, C99 extension. 181
isnormal, C99 extension 181
ISO/ANSI C
compiler extensions 133
specifying strict usage 124
iso646.h (library header file). 179
isunordered, C99 extension. 181
iswblank, C99 extension 182
italic style, in this guide xxii
I/O debugging, support for 59
I/O module, overriding in runtime library 44

K

keep_definition (pragma directive) 197
Kernighan & Ritchie function declarations 89
disallowing. 123
Kernighan, Brian W. xxii
keywords, extended. 8

L

-l (compiler option). 69, 114
labels. 141
assembler, making public. 122
__program_start. 48
Labrosse, Jean J. xxii
language extensions
descriptions 133
enabling 159
enabling (-e). 112
language overview 3
language (pragma directive) 159
libraries
building DLIB 38
definition of 4
runtime. 40
library configuration files
DLIB 39
Dlib_defaults.h. 46
dls08*libname*.h. 46
modifying 47
specifying 111
library documentation. 177
library functions 177
reference information. xxi
summary, DLIB 179
library header files 177
library modules
creating 115
overriding. 44
library object files. 177
library options, setting 8
library project template. 7, 46
--library_module (compiler option) 115
lightbulb icon, in this guide. xxiii
limits.h (library header file) 179
__LINE__ (predefined symbol) 173
linker command files. 26
customizing 26
using the -P command 28
using the -Z command 27
linker map file. 36
linker segment. *See* segment
linking
from the command line 4
required input. 4
listing, generating 114
literals, compound. 137
literature, recommended xxi
__LITTLE_ENDIAN__ (predefined symbol). 173
llabs, C99 extension 181
lldiv, C99 extension 181
local variables, *See* auto variables

locale support 54
adding 56
changing at runtime 56
removing 56
locale.h (library header file) 179
located data segments 34
location (pragma directive) 80, 160
LOCFRAME (assembler directive) 74
long float (data type), synonym for double 140
loop overhead, reducing 118
loop unrolling (compiler transformation) 85
disabling 118
loop-invariant expressions 86
low-level processor operations 133, 167
accessing 65
__low_level_init 48
customizing 50
low_level_init.c 47
__lseek (library function) 54

M

macros, variadic 175
main (function), definition 192
malloc (library function)
See also heap 17
implementation-defined behavior 200
Mann, Bernhard xxii
map, linker 36
math.h (library header file) 179
added C functionality 180
MATH_ERREXCEPT, C99 extension 181
math_errhandling, C99 extension 181
MATH_ERRNO, C99 extension 181
memory
accessing 12
dynamic 17
heap 17
non-initialized 91
RAM, saving 88
stack 16
saving 88
used by global or static variables 11
memory attributes for functions 20
memory layout, S08 11
memory placement
using pragma directive 14
using type definitions 14, 145
memory segment. *See* segment
memory types 12
pointers 14
specifying 13
structures 15
summary 13
memory (pragma directive) 197
message (pragma directive) 160
messages
disabling 123
forcing 160
--mfc (compiler option) 115
MISRA C, using . . . . . xxi, 104
--misrac_verbose (compiler option) 104
--misrac1998 (compiler option) 104
module consistency 61
rtmodel 163
module map, in linker map file 36
module name, specifying 115
module summary, in linker map file 36
--module_name (compiler option) 115
module_name (pragma directive) 197
__monitor (extended keyword) 90, 149
monitor functions 23, 149
multibyte character support 112
multi-file compilation 83

N

names block (call frame information) 75

naming conventions . . . xxiii
NAN, C99 extension . . . 181
NDEBUG (preprocessor symbol) . . . 174
non-banked functions, placing . . . 73
non-initialized variables, hints for . . . 91
non-scalar parameters, avoiding . . . 88
__non_banked (extended keyword) . . . 150
__non_banked (function pointer) . . . 128
NOP (assembler instruction) . . . 169
__noreturn (extended keyword) . . . 150
Normal DLIB (library configuration) . . . 39
Not a number (NaN) . . . 128
--no_code_motion (compiler option) . . . 116
--no_cse (compiler option) . . . 116
__no_init (extended keyword) . . . 91, 150
--no_inline (compiler option) . . . 117
__no_operation (intrinsic function) . . . 169
--no_path_in_file_macros (compiler option) . . . 117
no_pch (pragma directive) . . . 197
--no_typedefs_in_diagnostics (compiler option) . . . 118
--no_unroll (compiler option) . . . 118
--no_warnings (compiler option) . . . 119
--no_wrap_diagnostics (compiler option) . . . 119
NULL (macro), implementation-defined behavior . . . 198

O

-O (compiler option) . . . 119
-o (compiler option) . . . 120
object attributes . . . 146
object filename, specifying . . . 120–121
object module name, specifying . . . 115
object_attribute (pragma directive) . . . 92, 161
--omit_types (compiler option) . . . 120
once (pragma directive) . . . 198
--only_stdout (compiler option) . . . 121
__open (library function) . . . 54
operators
  @ . . . 80, 134
optimization
  code motion, disabling . . . 116
  common sub-expression elimination, disabling . . . 116
  configuration . . . 6
  disabling . . . 85
  function inlining, disabling (--no_inline) . . . 117
  hints . . . 87
  loop unrolling, disabling . . . 118
  specifying (-O) . . . 119
  summary . . . 84
  techniques . . . 85
  type-based alias analysis . . . 86
    disabling . . . 117
  using inline assembler code . . . 67
  using pragma directive . . . 161
optimization levels . . . 84
optimize (pragma directive) . . . 161
option parameters . . . 101
options, compiler. *See* compiler options
Oram, Andy . . . xxi
--output (compiler option) . . . 121
output files, from XLINK . . . 5
output format, default . . . 5
output (linker), specifying . . . 5
output (preprocessor) . . . 122
overhead, reducing . . . 85–86

P

paged code, *see* banking
parameters
  function . . . 71
  non-scalar, avoiding . . . 88
  register . . . 71
  rules for specifying a file or directory . . . 102
  specifying . . . 103
  stack . . . 71
  typographic convention . . . xxii
part number, of this guide . . . ii

performance of banked code . . . 21
permanent registers . . . 71
perror (library function),
implementation-defined behavior . . . 200
placement
  code and data . . . 183
  in named segments . . . 81
pointer types
  choosing . . . 78
  differences between . . . 14
  mixing . . . 140
pointers
  casting . . . 15, 129
  data . . . 129
  function . . . 128
  implementation-defined behavior . . . 195
porting, code containing pragma directives . . . 154
PPAGE (special function register) . . . 72
  supporting banked code . . . 3
_Pragma (predefined symbol) . . . 174
pragma directives . . . 8
  summary . . . 153
  bitfields . . . 127
  for absolute located data . . . 80
  list of all recognized . . . 197
  type_attribute, using . . . 14
predefined symbols
  overview . . . 8
  summary . . . 172
--preinclude (compiler option) . . . 121
--preprocess (compiler option) . . . 122
preprocessor
  output . . . 122
  overview . . . 171
preprocessor directives,
implementation-defined behavior of . . . 196
preprocessor extensions
  #warning message . . . 174
  __VA_ARGS__ . . . 175
preprocessor symbols . . . 172
  defining . . . 107
preserved registers . . . 71
__PRETTY_FUNCTION__ (predefined symbol) . . . 173
primitives, for special functions . . . 22
print formatter, selecting . . . 43
printf (library function)
  choosing formatter . . . 42
  configuration symbols . . . 52
  implementation-defined behavior . . . 199
processor configuration . . . 5
processor operations
  accessing . . . 65
  low-level . . . 133, 167
program entry label . . . 48
programming hints . . . 87
  banked systems . . . 21
__program_start (label) . . . 48
projects, basic settings for . . . 5
prototypes, enforcing . . . 123
ptrdiff_t (integer type) . . . 129
PUBLIC (assembler directive) . . . 122
publication date, of this guide . . . ii
--public_equ (compiler option) . . . 122
public_equ (pragma directive) . . . 198
putenv (library function), absent from DLIB . . . 57

Q

QCCS08 (environment variable) . . . 96
qualifiers, implementation-defined behavior . . . 196

R

-r (compiler option) . . . 122
raise (library function), configuring support for . . . 57
raise.c . . . 58
RAM memory, saving . . . 88
range errors, in linker . . . 35

RCODE (segment) 34, 189
__read (library function) 54
customizing 51
read formatter, selecting 44
reading guidelines xix
reading, recommended xxi
realloc (library function)
implementation-defined behavior 200
See also heap 17
recursive functions
avoiding 88
storing data on stack 17
reentrancy (DLIB) 178
reference information, typographic convention xxii
register parameters 71
registered trademarks ii
registers
callee-save, stored on stack 16
for function returns 72
implementation-defined behavior 195
in assembler-level routines 70
PPAGE, for banked code 3
preserved 71
scratch 71
virtual 70
remark (diagnostic message)
classifying 109
enabling 123
--remarks (compiler option) 123
remarks (diagnostic message) 99
remove (library function) 54
implementation-defined behavior 199
rename (library function) 54
implementation-defined behavior 199
__ReportAssert (library function) 58
required (pragma directive) 162
--require_prototypes (compiler option) 123
return addresses 72
return values, from functions 72
Ritchie, Dennis M. xxii
__root (extended keyword) 151
routines, time-critical 65, 133, 167
RTMODEL (assembler directive) 62
rtmodel (pragma directive) 163
__rt_version (runtime model attribute) 62
runtime environment 37
setting options 8
runtime libraries 40
choosing 7
introduction 177
choosing 41
customizing without rebuilding 42
naming convention 41
overriding modules in 44
runtime model attributes 61
runtime model definitions 163

S

scanf (library function)
choosing formatter 43
configuration symbols 52
implementation-defined behavior 200
scratch registers 71
section (pragma directive) 198
segment group name 29
segment map, in linker map file 36
segment memory types, in XLINK 26
segment names
declaring 164
suffixes 29
segment (pragma directive) 164
segments 183
code 34
data 28
definition of 25
initialized data 31
introduction 25

located data 34
naming 30
packing in memory 28
placing in sequence 27
static memory 29
summary 183
too long for address range 35
too long in linker 35
__segment_begin (extended operator). 135
__segment_end (extended operator) 135
SEI (assembler instruction). 168
semaphores
C example 23
operations on 149
setjmp.h (library header file). 179
setlocale (library function) 56
settings, basic for project configuration 5
__set_interrupt_state (intrinsic function) 169
severity level, of diagnostic messages. 99
specifying 100
SFR (special function registers) 90
shared object. 98
short (data type) 126
signal (library function)
configuring support for 57
implementation-defined behavior. 199
signal.c 58
signal.h (library header file) 179
signbit, C99 extension. 181
signed char (data type) 126–127
specifying 106
signed int (data type). 126
signed long (data type) 126
signed short (data type). 126
signed values, avoiding 77
--silent (compiler option) 123
silent operation, specifying 123
size_t (integer type) 129
skeleton code, creating for assembler language interface . . 68
skeleton.s78 (assembler source output). 69
Small code model 19
snprintf, C99 extension 181
__software_interrupt (intrinsic function) 169
source files, list all referred. 113
special function registers (SFR) 90
special function types 22
overview 9
sprintf (library function), choosing formatter 42
sscanf (library function), choosing formatter 43
stack 16, 32
advantages and problems using 17
changing default size of 32
cleaning after function return 72
contents of 16
internal data 185
layout. 71
saving space. 88
size. 33
stack parameters 71
stack pointer 16
stack segment, placing in memory 32
standard error 121
standard input 51
standard output 51
specifying 121
startup code
placement of 34
See also RCODE
startup, system 47
statements, implementation-defined behavior 196
static data, in linker command file 31
static memory segments 29
static overlay. 74
static variables
initialization. 31
taking the address of 87
stdarg.h (library header file) 179

stdbool.h (library header file) 126, 179
added C functionality 181
__STDC__ (predefined symbol) 173
STDC (pragma directive) 198
__STDC_VERSION__ (predefined symbol) 173
stddef.h (library header file) 127, 179
stderr 54, 121
stdin 54
implementation-defined behavior 199
stdint.h (library header file) 179
stdint.h, added C functionality 181
stdio.h (library header file) 179
stdio.h, additional C functionality 181
stdlib.h (library header file) 179
stdlib.h, additional C functionality 181
stdout 54, 121
implementation-defined behavior 199
Steele, Guy L. xxi
__stop (intrinsic function) 169
STOP (assembler instruction) 169
strerror (library function)
implementation-defined behavior 201
--strict_ansi (compiler option) 124
string.h (library header file) 179
strtod (library function), configuring support for 58
strtod, in stdlib.h 182
strtof, C99 extension 181
strtold, C99 extension 181
strtoll, C99 extension 181
strtoull, C99 extension 181
structure types, layout of 130
structures
anonymous 78, 135
implementation-defined behavior 195
incomplete arrays as last element 137
placing in memory type 15
subnormal numbers 128
__SUBVERSION__ (predefined symbol) 173
support, technical 100
SWI (assembler instruction) 169
symbols
anonymous, creating 137
including in output 162
listing in linker map file 36
overview of predefined 8
preprocessor, defining 107
syntax
compiler options 101
extended keywords 13, 144–146
system startup 47
customizing 50
system termination 49
C-SPY interface to 50
system (library function)
configuring support for 57
implementation-defined behavior 200
system_include (pragma directive) 198
S08
memory layout 11
supported devices 3

T

__task (extended keyword) 151
technical support, IAR Systems 100
Terminal I/O window, making available 60
terminal output, speeding up 60
termination, of system 49
32-bits (floating-point format) 127
__TIME__ (predefined symbol) 173
time zone (library function)
implementation-defined behavior 201
time (library function), configuring support for 58
time-critical routines 65, 133, 167
time.c 58
time.h (library header file) 179
Tiny DLIB (library configuration) 39
tips, programming 87
tools icon, in this guide xxii

trademarks . . . ii
transformations, compiler . . . 83
translation, implementation-defined behavior . . . 191
type attributes . . . 143
  specifying . . . 165
type definitions, used for specifying memory storage . 14, 145
type information, omitting . . . 120
type qualifiers, const and volatile . . . 131
typedefs
  excluding from diagnostics . . . 118
  repeated . . . 140
type-based alias analysis (compiler transformation) . . . 86
  disabling . . . 117
type_attribute (pragma directive) . . . 14, 165
typographic conventions . . . xxii

U

uintptr_t (integer type) . . . 130
underflow range errors,
implementation-defined behavior . . . 198
unions
  anonymous . . . 78, 135
  implementation-defined behavior . . . 195
unsigned char (data type) . . . 126–127
  changing to signed char . . . 106
unsigned int (data type) . . . 126
unsigned long (data type) . . . 126
unsigned short (data type) . . . 126

V

variable type information, omitting in object output . . . 120
variables
  auto . . . 16
  defined inside a function . . . 16
  global, placement in memory . . . 11
  hints for choosing . . . 87
  local. *See* auto variables
  non-initialized . . . 91
  omitting type info . . . 120
  placing at absolute addresses . . . 81
  placing in named segments . . . 81
  static
    placement in memory . . . 11
    taking the address of . . . 87
  static and global, initializing . . . 31
vector (pragma directive) . . . 22, 165
__VER__ (predefined symbol) . . . 173
version, IAR Embedded Workbench . . . ii
version, of compiler . . . 173
vfscanf, C99 extension . . . 181
vfwscanf, C99 extension . . . 182
virtual registers . . . 70
void, pointers to . . . 140
volatile (keyword) . . . 89
volatile, declaring objects . . . 131
vscanf, C99 extension . . . 181
vsnprintf, C99 extension . . . 181
vsscanf, C99 extension . . . 181
vswscanf, C99 extension . . . 182
vwscanf, C99 extension . . . 182

W

WAIT (assembler instruction) . . . 169
__wait_for_interrupt (intrinsic function) . . . 169
#warning message (preprocessor extension) . . . 174
warnings . . . 99
  classifying . . . 110
  disabling . . . 119
  exit code . . . 124
warnings icon, in this guide . . . xxiii
warnings (pragma directive) . . . 198
--warnings_affect_exit_code (compiler option) . . . 98
--warnings_are_errors (compiler option) . . . 124
wchar.h (library header file) . . . 179
wchar.h, added C functionality . . . 182

wchar_t (data type), adding support for in C . . . 127
wcstof, C99 extension . . . 182
wcstolb, C99 extension . . . 182
wctype.h (library header file) . . . 179
wctype.h, added C functionality . . . 182
web sites, recommended . . . xxii
__write (library function) . . . 54
customizing . . . 51

X

XLINK errors
range error . . . 35
segment too long . . . 35
XLINK output files . . . 5
XLINK segment memory types . . . 26
xreportassert.c . . . 58

Symbols

#include files, specifying . . . 96, 113
#warning message (preprocessor extension) . . . 174
-D (compiler option) . . . 107
-e (compiler option) . . . 112
-f (compiler option) . . . 113
-I (compiler option) . . . 113
-l (compiler option) . . . 69, 114
-O (compiler option) . . . 119
-o (compiler option) . . . 120
-r (compiler option) . . . 122
--char_is_signed (compiler option) . . . 106
--code_model (compiler option) . . . 106
--core (compiler option) . . . 106
--debug (compiler option) . . . 107
--dependencies (compiler option) . . . 108
--diagnostics_tables (compiler option) . . . 110
--diag_error (compiler option) . . . 109
--diag_remark (compiler option) . . . 109
--diag_suppress (compiler option) . . . 110
--diag_warning (compiler option) . . . 110
--discard_unused_publics (compiler option) . . . 111
--dlib_config (compiler option) . . . 111
--enable_multibytes (compiler option) . . . 112
--error_limit (compiler option) . . . 112
--header_context (compiler option) . . . 113
--library_module (compiler option) . . . 115
--mfc (compiler option) . . . 115
--misrac_verbose (compiler option) . . . 104
--misrac1998 (compiler option) . . . 104
--module_name (compiler option) . . . 115
--no_code_motion (compiler option) . . . 116
--no_cross_call (compiler option) . . . 116
--no_cse (compiler option) . . . 116
--no_inline (compiler option) . . . 117
--no_path_in_file_macros (compiler option) . . . 117
--no_tbaa (compiler option) . . . 117
--no_typedefs_in_diagnostics (compiler option) . . . 118
--no_unroll (compiler option) . . . 118
--no_warnings (compiler option) . . . 119
--no_wrap_diagnostics (compiler option) . . . 119
--omit_types (compiler option) . . . 120
--only_stdout (compiler option) . . . 121
--output (compiler option) . . . 121
--preinclude (compiler option) . . . 121
--preprocess (compiler option) . . . 122
--remarks (compiler option) . . . 123
--require_prototypes (compiler option) . . . 123
--silent (compiler option) . . . 123
--strict_ansi (compiler option) . . . 124
--warnings_affect_exit_code (compiler option) . . . 98, 124
--warnings_are_errors (compiler option) . . . 124
@ (operator) . . . 80, 134
_Exit (library function) . . . 49
_exit (library function) . . . 49
_Exit, C99 extension . . . 181
_Pragma (predefined symbol) . . . 174
__ALIGNOF__ (operator) . . . 135
__asm (language extension) . . . 136

__banked (extended keyword) 20, 147
__banked (function pointer) 128
__BASE_FILE__ (predefined symbol). 172
__BGND (intrinsic function) 168
__BUILD_NUMBER__ (predefined symbol) 172
__close (library function) 54
__code_model (runtime model attribute) 62
__CODE_MODEL__ (predefined symbol). 172
__CORE__ (predefined symbol). 172
__data8 (extended keyword) 129, 148
__data16 (extended keyword) 129, 148
__DATE__ (predefined symbol) 172
__disable_interrupt (intrinsic function). 168
__enable_interrupt (intrinsic function) 168
__exit (library function) 49
__FILE__ (predefined symbol). 172
__FUNCTION__ (predefined symbol) 142, 172
__func__ (predefined symbol) 142, 172
__gets, in stdio.h. 181
__get_interrupt_state (intrinsic function) 168
__IAR_SYSTEMS_ICC__ (predefined symbol) 172
__ICCS08__ (predefined symbol). 172
__illegal_opcode (intrinsic function) 169
__interrupt (extended keyword) 22, 149
using in pragma directives 165
__intrinsic (extended keyword). 149
__LINE__ (predefined symbol) 173
__LITTLE_ENDIAN__ (predefined symbol). 173
__low_level_init 48
__low_level_init, customizing 50
__lseek (library function) 54
__monitor (extended keyword) 90, 149
__non_banked (extended keyword) 20, 150
__non_banked (function pointer) 128
__noreturn (extended keyword) 150
__no_init (extended keyword) 91, 150
__no_operation (intrinsic function). 169
__open (library function) 54
__PRETTY_FUNCTION__ (predefined symbol). 173
__printf_args (pragma directive). 162, 198
__program_start (label). 48
__qsort (library function) 182
__read (library function). 54
customizing 51
__ReportAssert (library function). 58
__root (extended keyword) 151
__rt_version (runtime model attribute) 62
__scanf_args (pragma directive) 164, 198
__segment_begin (extended operator 135
__segment_end (extended operators) 135
__set_interrupt_state (intrinsic function) 169
__software_interrupt (intrinsic function) 169
__STDC_VERSION__ (predefined symbol) 173
__STDC__ (predefined symbol). 173
__stop (intrinsic function). 169
__SUBVERSION__ (predefined symbol). 173
__task (extended keyword) 151
__TIME__ (predefined symbol) 173
__ungetchar, in stdio.h 181
__VA_ARGS__ (preprocessor extension). 175
__VER__ (predefined symbol) 173
__wait_for_interrupt (intrinsic function). 169
__write (library function) 54
customizing 51
__write_array, in stdio.h 181
__write_buffered (DLIB library function). 60

Numerics

32-bits (floating-point format) 127